



INVESTOR IN PEOPLE

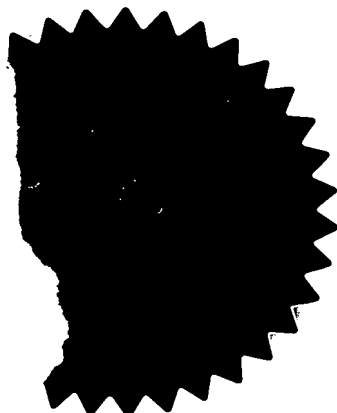
The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

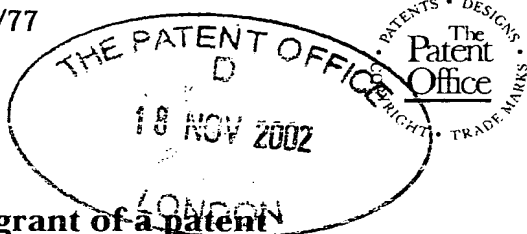
In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.



Signed

Dated 5 November 2003



177
19NOV02 E764402-1 002246
P01/7700 0.00-0226875.3

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

The Patent Office

Cardiff Road
Newport
South Wales
NP10 8QQ

1. Your reference

P015381GB

18 NOV 2002

2. Patent application number

(The Patent Office will fill in this part)

0226875.3

3. Full name, address and postcode of the or of each applicant (underline all surnames)

ARM Limited
110 Fulbourn Road
Cherry Hinton
Cambridge
CB1 9NJ

Patents ADP number (if you know it)

7498124003

If the applicant is a corporate body, give the country/state of its incorporation

United Kingdom

4. Title of the invention

CONTROL OF ACCESS TO A MEMORY BY A DEVICE

5. Name of your agent (if you have one)

D Young & Co

"Address for service" in the United Kingdom to which all correspondence should be sent (including the postcode)

21 New Fetter Lane
London
EC4A 1DA

Patents ADP number (if you know it)

59006

6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and (if you know it) the or each application number

Country

Priority application number
(if you know it)

Date of filing
(day / month / year)

7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application

Number of earlier application

Date of filing
(day / month / year)

8. Is a statement of inventorship and of right to grant of a patent required in support of this request? (Answer 'Yes' if:

Yes

- a) any applicant named in part 3 is not an inventor, or
 - b) there is an inventor who is not named as an applicant, or
 - c) any named applicant is a corporate body.
- See note (d))

CONTROL OF ACCESS TO A MEMORY BY A DEVICE

5 The present invention concerns techniques for controlling access to a memory by a device.

A data processing apparatus will typically include a processor for running applications loaded onto the data processing apparatus. The processor will operate under the control of an operating system. The data required to run any particular application will typically be stored within a memory of the data processing apparatus. It
10 will be appreciated that the data may consist of the instructions contained within the application and/or the actual data values used during the execution of those instructions on the processor.

There arise many instances where the data used by at least one of the applications is sensitive data that should not be accessible by other applications that can
15 be run on the processor. An example would be where the data processing apparatus is a smart card, and one of the applications is a security application which uses sensitive data, such as for example secure keys, to perform validation, authentication, decryption and the like. It is clearly important in such situations to ensure that such sensitive data is kept secure so that it cannot be accessed by other applications that may be loaded onto
20 the data processing apparatus, for example hacking applications that have been loaded onto the data processing apparatus with the purpose of seeking to access that secure data.

In known systems, it has typically been the job of the operating system developer to ensure that the operating system provides sufficient security to ensure that the secure
25 data of one application cannot be accessed by other applications running under the control of the operating system. However, as systems become more complex, the general trend is for operating systems to become larger and more complex, and in such situations it becomes increasingly difficult to ensure sufficient security within the operating system itself.

30 Examples of systems seeking to provide secure storage of sensitive data and to provide protection against malicious program code are those described in United States

Hence, the partition checking logic is arranged to police accesses to the memory in order to ensure that no accesses to the secure memory occur when the device is operating in a non-secure mode.

The partition checking logic will have access to information about the partitions
5 between the secure memory and the non-secure memory. It will be appreciated that this partitioning information could be hardwired in embodiments where the physical partition between secure memory and non-secure memory cannot be altered. However, in preferred embodiments, the partitioning between secure memory and non-secure memory is configurable by the device when the device is operating in a predetermined
10 secure mode, and in such embodiments, the partition checking logic is managed by the device when operating in that predetermined secure mode. Hence, if a rogue application were installed onto the device with the aim of seeking to access secure data, that application will not be able to be run in the secure domain, and accordingly it cannot alter the partitioning information. Hence, even if that application can output a memory
15 access request which is seeking to access a location within the secure memory, the partition checking logic will detect that that application executing in a non-secure mode of the device is seeking to access a secure memory location, and will prevent the access taking place.

It will be appreciated that there are a number of different ways in which the
20 partition checking logic might receive information from the device about which domain that device is operating in. However, in preferred embodiments the memory access request issued by the device includes a domain signal identifying whether the device is operating in said at least one secure mode or said at least one non-secure mode.

This domain signal hence identifies whether the device is operating in the secure
25 domain or the non-secure domain, and thus the partition checking logic can be triggered to check the memory access request in the event that that domain signal indicates that the device is operating in the non-secure domain. In preferred embodiments, the partition checking logic does not perform any partition checking when the device is operating in the secure domain, since in preferred embodiments the device can access
30 both the secure memory and the non-secure memory when operating in a secure mode.

coupled to the bus, each one that can operate in both secure and non-secure modes may independently control the partition checking logic when that device is operating in a secure mode, in such situations the partition checking logic having access to partitioning information specific to each separate device. However, preferably, one of the devices is
5 responsible for management of the partition checking logic.

In preferred embodiments, at least one device is a chip incorporating a processor, the chip further comprising a memory management unit operable, when the processor generates the memory access request, to perform one or more predetermined access control functions to control issuance of the memory access request onto the device bus.

10 It will be appreciated that the memory coupled to the device bus can take a variety of forms, for example Random Access Memory (RAM), Read Only Memory (ROM), a hard disk, registers within certain peripheral devices, etc. In addition to such memory, it will also be appreciated that when the device is a chip incorporating a processor, there may also be certain on-chip memory within the chip, for example, cache
15 memory, Tightly Coupled Memory (TCM), etc.

Hence, in certain embodiments, the chip further comprises: on-chip memory coupled to the processor via a system bus, the on-chip memory operable to store data required by the processor, the on-chip memory comprising secure on-chip memory for storing secure data and non-secure on-chip memory for storing non-secure data; and
20 on-chip partition checking logic coupled to the system bus and operable whenever the memory access request is generated by the processor when operating in said non-secure mode to detect if the memory access request is seeking to access either the secure memory or the secure on-chip memory, and upon such detection to prevent the access specified by that memory access request.

25 Since the partition checking logic coupled to the device bus cannot perform any partition checking function with regard to memory internal to the chip, on-chip partition checking logic is provided in such embodiments to ensure that when the processor is operating in a non-secure mode, it cannot access any part of the secure memory system, whether that be parts of the secure memory coupled to the device bus, or parts of the on-
30 chip memory that contains secure data.

Figure 6 illustrates one programmer's model of a register bank of a processor depending upon the processing mode;

Figure 7 illustrates an example of providing separate register banks for a
5 secure domain and a non-secure domain;

Figure 8 schematically illustrates a plurality of processing modes with switches between security domains being made via a separate monitor mode;

10 Figure 9 schematically illustrates a scenario for security domain switching using a mode switching software interrupt instruction;

Figure 10 schematically illustrates one example of how non-secure interrupt requests and secure interrupt requests may be processed by the system;
15

Figures 11A and 11B schematically illustrate an example of non-secure interrupt request processing and an example of secure interrupt request processing in accordance with Figure 10;

20 Figure 12 illustrates an alternative scheme for the handling of non-secure interrupt request signals and secure interrupt request signals compared to that illustrated in Figure 10;

Figures 13A and 13B illustrate example scenarios for dealing with a non-secure interrupt request and a secure interrupt request in accordance with the scheme
25 illustrated in Figure 12;

Figure 14 is an example of a vector interrupt table;

30 Figure 15 schematically illustrates multiple vector interrupt tables associated with different security domains;

Figure 26 is a diagram illustrating the use of both a non-secure page table and a secure page table in preferred embodiments of the present invention;

Figure 27 is a diagram illustrating two forms of flag used within the main translation lookaside buffer (TLB) of preferred embodiments;

Figure 28 illustrates how memory may be partitioned after a boot stage in one embodiment of the present invention;

Figure 29 illustrates the mapping of the non-secure memory by the memory management unit following the performance of the boot partition in accordance with an embodiment of the present invention;

Figure 30 illustrates how the rights of a part of memory can be altered to allow a secure application to share memory with a non-secure application in accordance with an embodiment of the present invention;

Figure 31 illustrates how devices may be connected to the external bus of the data processing apparatus in accordance with one embodiment of the present invention;

Figure 32 is a block diagram illustrating how devices may be coupled to the external bus in accordance with the second embodiment of the present invention;

Figure 33 schematically shows possible granularity of monitoring functions for different modes and applications running on a processor;

Figure 34 shows possible ways of initiating different monitoring functions;

Figure 35 shows a table of control values for controlling availability of different monitoring functions;

defining which activities are to be traced. The trace signals are typically output to a trace buffer from where they can subsequently be analysed. A vectored interrupt controller 21 is provided for managing the servicing of a plurality of interrupts which may be raised by various peripherals ().

5

Further, as shown in Figure 1, another monitoring functionality that can be provided within the core 10 is a debug function, a debugging application external to the data processing apparatus being able to communicate with the core 10 via a Joint Test Access Group (JTAG) controller 18 which is coupled to one or more scan chains 12. Information about the status of various parts of the processor core 10 can be output via the scan chains 12 and the JTAG controller 18 to the external debugging application. An In Circuit Emulator (ICE) 20 is used to store within registers 24 conditions identifying when the debug functions should be started and stopped, and hence for example will be used to store breakpoints, watchpoints, etc.

15

The core 10 is coupled to a system bus 40 via memory management logic 30 which is arranged to manage memory access requests issued by the core 10 for access to locations in memory of the data processing apparatus. Certain parts of the memory may be embodied by memory units connected directly to the system bus 40, for example the Tightly Coupled Memory (TCM) 36, and the cache 38 illustrated in Figure 1. Additional devices may also be provided for accessing such memory, for example a Direct Memory Access (DMA) controller 32. Typically, various control registers 34 will be provided for defining certain control parameters of the various elements of the chip, these control registers also being referred to herein as coprocessor 15 (CP 15) registers.

25

The chip containing the core 10 will typically be coupled to an external bus 70 (for example a bus operating in accordance with the "Advanced Microcontroller Bus Architecture" (AMBA) specification developed by ARM Limited) via an external bus interface 42, and various devices may be connected to the external bus 70. These devices may include master devices such as a digital signal processor (DSP) 50, or a

30

operation with the non-secure operating system 74. In the secure domain, a secure kernel program 80 is provided. The secure kernel program 80 can be considered to form a secure operating system. Typically such a secure kernel program 80 will be designed to provide only those functions which are essential to processing activities which must be provided in the secure domain such that the secure kernel 80 can be as small and simple as possible since this will tend to make it more secure. A plurality of secure applications 82, 84 are illustrated as executing in combination with the secure kernel 80.

Figure 3 illustrates a matrix of processing modes associated with different security domains. In this particular example the processing modes are symmetrical with respect to the security domain and accordingly Mode 1 and Mode 2 exist in both secure and non-secure forms.

The monitor mode has the highest level of security access in the system and is the only mode entitled to switch the system between the non-secure domain and the secure domain in either direction. Thus, all domain switches take place via a switch to the monitor mode and the execution of the monitor program 72 within the monitor mode.

Figure 4 schematically illustrates another set of non-secure domain processing modes 1, 2, 3, 4 and secure domain processing modes a, b, c. In contrast to the symmetric arrangement of Figure 3, Figure 4 shows that some of the processing modes may not be present in one or other of the security domains. The monitor mode 86 is again illustrated as straddling the non-secure domain and the secure domain. The monitor mode 86 can be considered a secure processing mode, since the secure status flag may be changed in this mode and monitor program 72 in the monitor mode has the ability to itself set the security status flag it effectively provides the ultimate level of security within the system as a whole.

fast and efficient mechanism of placing it in a register which is accessible in both the non-secure domain and the secure domain.

An important advantage of having secure register bank is to avoid the need for
5 flushing the contents of registers before switching from one world to the other. If
latency is not a critical issue, a simpler hardware system with no duplicated registers
for the secure domain world may be used, e.g. Figure 6. The monitor mode is
responsible switching from one domain to the other. Restoring context, saving
previous context, as well as flushing registers is performed by a monitor program at
10 least partially executing in monitor mode. The system behaves thus like a
virtualisation model. This type of embodiment is discussed further below. Reference
should be made to, for example, the programmer's model of the ARM7 upon which
the security features described herein build.

15 *Processor Modes*

Instead of duplicating modes in secure world, the same modes support both
secure and non-secure domains (see Figure 8). Monitor mode is aware of the current
status of the core, either secure or non-secure (e.g. as read from an S bit stored in a
20 coprocessor configuration register).

In the Figure 8, whenever an SMI (Software Monitor Interrupt instruction)
occurs, the core enters monitor mode to switch properly from one world to the other.

25 With reference to Figure 9:

1. The scheduler launches thread 1
2. Thread 1 needs to perform a secure function => SMI secure call, the core enters
monitor mode. Under hardware control the current PC and CPSR (current processor
30 status register) are stored in R14_mon and SPSR_mon (saved processor status register
for the monitor mode) and IRQ/FIQ interrupts are disabled.

terminated an SMI to go to monitor mode may be executed and the monitor can switch to the OS in non-secure world (non-secure svc mode) if desired. If it is desired to use a legacy OS this can simply boot in secure supervisor mode and ignore the secure state.

5 SMI INSTRUCTION

This instruction (a mode switching software interrupt instruction) can be called from any non-secure modes in the non-secure domain (as previously mentioned it may be desired to restrict SMIs to privileged modes), but the target entry point determined by the associated vector is always fixed and within monitor mode. Its up to the SMI handler to branch to the proper secure function that must be run (e.g. controlled by an operand passed with the instruction).

Passing parameters from non-secure world to secure world can be performed using the shared registers of the register bank within a Figure 6 type register bank.

15

When a SMI occurs in non-secure world, the ARM core may do the following actions in hardware:

- Branch to SMI vector (in secure memory access is allowed since you will now be in monitor mode) into monitor mode
- 20 - Save PC into R14_mon and CPSR into SPSR_mon
- Set the S bit using the monitor program
- Start to execute secure exception handler in monitor mode (restore/save context in case of multi-threads)
- Branch to secure user mode (or another mode, like svc mode) to execute the appropriate function
- 25 - IRQ and FIQ are disabled while the core is in monitor mode (latency is increased)

Secure World Exit

30 There are two possibilities to exit secure world:

In this solution, two distinct pins are required to support secure and non-secure interrupts.

While in Non Secure world, if

- 5 - an IRQ occurs, the core goes to IRQ mode to handle this interrupt as in ARM cores such as the ARM7
- a SIRQ occurs, the core goes to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

10 ***While in Secure world, if***

- an SIRQ occurs, the core goes to the secure IRQ handler. The core does not leave the secure world
- an IRQ occurs, the core goes to monitor mode where secure context is saved, then to a non-secure IRQ handler to deal with this non-secure interrupt.

15

In other words, when an interrupt that does not belong to the current world occurs, the core goes directly to monitor mode, otherwise it stays in the current world (see Figure 10).

20 **IRQ Occurring in Secure World**

See Figure 11A:

1. The scheduler launches thread 1.
- 25 2. Thread 1 needs to perform a secure function => SMI secure call, the core enters monitor mode. Current PC and CPSR are stored in R14_mon and SPSR_mon, IRQ/FIQ are disabled.
3. The monitor handler (program) does the following tasks:
 - The S bit is set.
 - 30 - Saves at least R14_mon and SPSR_mon in a stack (and possibly other registers are also pushed) so that non-secure context cannot be lost if an exception occurs whilst the secure application is running.

See Figure 11B:

1. The schedule launches thread 1
2. A SIRQ occurs while secure thread 1 is running. The core jumps directly to monitor mode (specific vector) and stores current PC in R14_mon and CPSR in SPSR_mon in monitor mode, IRQ/FIQ are then disabled.
3. Non-Secure context must be saved, then the core goes to a secure IRQ handler.
4. The IRQ handler services the SIRQ, then gives control back to the monitor mode handler using an SMI with appropriate parameters.
5. The monitor handler restores non-secure context so that a SUBS instruction makes the core return to the non-secure world and resumes the interrupted thread 1.
6. Thread 1 executes until the end, then gives the hand back to the scheduler.

The mechanism of Figure 11A has the advantage of providing a deterministic way to enter secure world. However, there are some problems associated with interrupt priority: e.g. while a SIRQ is running in secure interrupt handler, a non-secure IRQ with higher priority may occur. Once the non-secure IRQ is finished, there is a need to recreate the SIRQ event so that the core can resume the secure interrupt.

Solution Two

In this mechanism (See Figure 12) two distinct pins, or only one, may support secure and non-secure interrupts. Having two pins reduces interrupt latency.

While in Non Secure world, if

- an IRQ occurs, the core goes to IRQ mode to handle this interrupt like in ARM7 systems
 - a SIRQ occurs, the core goes to an IRQ handler where an SMI instruction will make the core branch to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

While in a Secure world, if

also save in these same registers the non-secure context that must be restored once the IRQ routine will be finished.

7. The IRQ handler services the IRQ, then gives control back to thread 1 in the non-secure world. By restoring SPRS_irq and R14_irq into the CPSR and PC, the core is now pointing onto the SMI instruction that has been interrupted.
8. The SMI instruction is re-executed (same instruction as 2).
9. The monitor handler sees this thread has previously been interrupted, and restores the thread 1 context. It then branches to secure thread 1 in user mode, pointing to the instruction that has been interrupted.
10. Secure thread 1 runs until it finishes, then branches onto the 'return from secure'; function in monitor mode (dedicated SMI).
11. The 'return from secure' function does the following tasks:
 - indicates that secure thread 1 is finished (i.e., in the case of a thread ID table, remove thread 1 from the table).
 - restores from stack non-secure context and flushes required registers, so that no secure information can be read once we return in non-secure world.
 - branches back to the non-secure world with a SUBS instruction, restoring the PC (from restored R14_mon) and CPSR (from SPSR_mon). The return point in the non-secure world should be the instruction following the previously executed SMI in thread 1.
11. Thread 1 executes until the end, then gives the hand back to the scheduler.

SIRQ Occurring in Non-Secure World

See Figure 13B:

1. The schedule launches thread 1.
2. A SIRQ occurs while secure thread 1 is running.
3. The core jumps directly irq mode and stores current PC in R14_irq and CPSR in SPSR_irq. IRQ is then disabled. The IRQ handler detects this is a SIRQ and performs a SMI instruction with appropriate parameters.

		mode
SWI	0x08	Supervisor mode/Monitor mode
Prefetch Abort	0x0C	Abort mode/Monitor mode
Data Abort	0x10	Abort mode/Monitor Mode
SMI	0x14	Undef mode/Monitor mode
IRQ/SIRQ	0x18	IRQ mode
FIQ	0x1C	FIQ mode

NB. The Reset entry is only in the secure vector table. When a Reset is performed in non secure world, the core hardware forces entry of supervisor mode and setting of the S bit so that the Reset vector can be accessed in secure memory.

Figure 15 illustrates three exception vector tables respectively applicable to a secure mode, a non-secure mode and the monitor mode. These exception vector tables may be programmed with exception vectors in order to match the requirements and characteristics of the secure and non-secure operating systems. Each of the exception vector tables may have an associated vector table base address register within CP15 storing a base address pointing to that table within memory. When an exception occurs the hardware will reference the vector table base address register corresponding to the current state of the system to determine the base address of the vector table to be used. Alternatively, the different virtual to physical memory mappings applied in the different modes may be used to separate the three different vector table sstored at different physical memory addresses. As illustrated in Figure 16, an exception trap mask register is provided in a system (configuration controlling) coprocessor (CP15) associated with the processor core. This exception trap mask register provides flags associated with respective exception types. These flags indicate whether the hardware should operate to direct processing to either the vector for the exception concerned within its current domain or should force a switch to the

handler, but this then execute instructions which direct processing to the secure exception handler or the monitor mode exception handler.

5

Figure 17 is a flow diagram schematically illustrating the operation of the system so as to support another possible type of switching request associated with a new type of exception. At step 98 the hardware detects any instruction which is attempting to change to monitor mode as indicate in a current program status register (CPSR).

10 When such an attempt is detected, then a new type of exception is triggered, this being referred to herein as a CPSR violation exception. The generation of this CPSR violation exception at step 100 results in reference to an appropriate exception vector within the monitor mode and the monitor program is run at step 102 to handle the CPSR violation exception.

15

It will be appreciated that the mechanisms for initiating a switch between secure domain and non-secure domain discussed in relation to Figure 17 may be provided in addition to support for the SMI instruction previously discussed. This exception mechanism may be provided to respond to unauthorised attempts to switch mode as
20 all authorised attempts should be made via an SMI instruction. Alternatively, such a mechanism may be legitimate ways to switch between the secure domain and the non-secure domain or may be provided in order to give backwards compatibility with existing code which, for example, might seek to clear the processing status register as part of its normal operation even though it was not truly trying to make an
25 unauthorised attempt to switch between the secure domain and the non-secure domain.

A description of an alternative embodiment(s) of the present technique considered from a programmer's model view is given below in relation to Figures 18 to 20 as
30 follows:

The concept of the Carbon architecture, which is the term used herein for processors using the present techniques, consists in separating two worlds, one secure and one non-secure. The secure world must not leak any data to non-secure world.

5 In the proposed solution, the secure and non-secure states will share the same (existing) register bank. As a consequence, all current modes present in ARM cores (Abort, Undef, Irq, User, ...) will exist in each state.

10 The core will know it operates in secure or non-secure state thanks to a new state bit, the S (secure) bit, instantiated in a dedicated CP15 register.

Controlling which instruction or event is allowed to modify the S bit, i.e. to change from one state to the other, is a key feature of the security of the system. The current solution proposes to add a new mode, the Monitor mode, that will “supervise”
15 switching between the two states. The Monitor mode, by writing to the appropriate CP15 register, would be the only one allowed to alter the S bit.

Finally, we propose to add some flexibility to the exception handling. All exceptions, apart from the reset, would be handled either in the state where they
20 happened, or would be directed to the Monitor mode. This would be left configurable thanks to a dedicated CP15 register.

The details of this solution are discussed in the following paragraphs.

25 **Processor state and modes**

Carbon new features

Secure or Non-secure state (S bit)

One major feature of the Carbon core is the presence of the S bit, which indicates whether the core is in a Secure (S=1) or Non-secure (S=0) state. When in
30 Secure state, the core would be able to access any data in the Secure or Non-secure

There may be a need for a dedicated CPSR violation exception. This exception would be raised by any attempt to switch to Monitor mode by directly writing the CPSR from any Non-secure mode or Secure user mode.

5

All exceptions except Reset are in effect disabled when Monitor mode is active:

- all interrupts are masked;
- all memory exceptions are either ignored or cause a fatal exception.
- 10 • undefined/SWI/SMI are ignored or cause a fatal exception.

When entering Monitor mode, the interrupts are automatically disabled and the system monitor should be written such that none of the other types of exception can happen while the system monitor is running.

15

Monitor mode needs to have some private registers. This solution proposes that we only duplicate the minimal set of registers, i.e R13 (sp_mon), R14 (lr_mon) and SPSR (spsr_mon).

20 In Monitor mode, the MMU will be disabled (flat address map) as well as the MPU or partition checker (the Monitor mode will always perform secure privileged external accesses). However, specially programmed MPU region attributes (cacheability, ...) would still be active.

25 **New instruction**

This proposal requires adding one new instruction to the existing ARM instruction set.

30 The SMI (Software Monitor Interrupt) instruction would be used to enter the Monitor mode, branching at a fixed SMI exception vector. This instruction would be

Another solution proposed involves duplicating all (or most of) the existing register bank, thus having two physically separated register banks between the Secure and Non-secure state. This solution has the main advantage of clearly separating the secure and non-secure data contained in the registers. It also allows fast context switching between the secure and non-secure states. However, the drawback is that passing parameters through registers becomes difficult, unless we create some dedicated instructions to allow the secure world access the non-secure registers

Figure 19 illustrates the available registers depending on the processor mode. Note that the processor state has no impact on this topic.

Exceptions

Secure interrupts

Current Solution

It is currently proposed to keep the same interrupt pins as in the current cores, i.e. IRQ and FIQ. In association with the Exception Trap Mask register (defined later in the document), there should be sufficient flexibility for any system to implement and handle different kind of interrupts.

VIC enhancement

We could enhance the VIC (Vectored Interrupt Controller) in the following way: the VIC may contain one Secure information bit associated to each vectored address. This bit would be programmable by the Monitor or Secure privileged modes only. It would indicate whether the considered interrupt should be treated as Secure, and thus should be handled on the Secure side.

We would also add two new Vector Address registers, one for all Secure Interrupts happening in Non-Secure state, the other one for all Non-Secure interrupts happening in Secure state.

Non-Secure Interrupt	The VIC has no Vector associated to this interrupt in the Secure domain. It thus presents to the core the address contained in the Vector address register dedicated to all Non-secure interrupts occurring in Secure world. The core, still in Secure world, then branches to this address, where it should find an SMI instruction to switch to Non-secure world. Once in Non-secure world, it would be able to have access to the correct ISR.	No need to switch between worlds. The VIC directly presents to the core the Non-secure address associated to the interrupt line. The core simply has to branch at this address where it should find the associated Non-secure ISR.
-----------------------------	---	--

Exception handling configurability

In order to improve Carbon flexibility, a new register, the Exception Trap

- 5 Mask, would be added in CP15. This register would contain the following bits:
- Bit 0: Undef exception (Non-secure state)
 - Bit 1: SWI exception (Non-secure state)
 - Bit 2: Prefetch abort exception (Non-secure state)
 - Bit 3: Data abort exception (Non-secure state)
 - 10 - Bit 4: IRQ exception (Non-secure state)
 - Bit 5: FIQ exception (Non-secure state)
 - Bit 6: SMI exception (both Non-secure/Secure states)
 - Bit 16: Undef exception (Secure state)
 - Bit 17: SWI exception (Secure state)
 - 15 - Bit 18: Prefetch abort exception (Secure state)
 - Bit 19: Data abort exception (Secure state)
 - Bit 20: IRQ exception (Secure state)
 - Bit 21: FIQ exception (Secure state)

0x04	Undef	Undef	Undefined instruction executed when core is in Non-Secure state and Exception Trap Mask reg [Non-secure Undef]=0
0x08	SWI	Supervisor	SWI instruction executed when core is in Non-Secure state and Exception Trap Mask reg [Non-secure SWI]=0
0x0C	Prefetch Abort	Abort	Aborted instruction when core is in Non-Secure state and Exception Trap Mask reg [Non-secure PAbort]=0
0x10	Data Abort	Abort	Aborted data when core is in Non-Secure state and Exception Trap Mask reg [Non-secure DAbort]=0
0x14	Reserved		
0x18	IRQ	IRQ	IRQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure IRQ]=0
0x1C	FIQ	FIQ	FIQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure FIQ]=0

In secure memory:

Address	Exception	Mode	Automatically accessed when
0x00	Reset*	Supervisor	Reset pin asserted
0x04	Undef	Undef	Undefined instruction executed when core is in Secure state and Exception Trap Mask reg [Secure Undef]=0
0x08	SWI	Supervisor	SWI instruction executed when core is in Secure state and Exception Trap Mask reg [Secure SWI]=0

0x0C	Prefetch Abort	Monitor	Aborted instruction when core is in Secure state and Exception Trap Mask reg [Secure IAbort]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure Iabort]=1
0x10	Data Abort	Monitor	Aborted data when core is in Secure state and Exception Trap Mask reg [Secure PAbort]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure Pabort]=1
0x14	SMI	Monitor	
0x18	IRQ	Monitor	- IRQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure IRQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure IRQ]=1
0x1C	FIQ	Monitor	- FIQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure FIQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure FIQ]=1

In Monitor mode, the exceptions vectors may be duplicated, so that each exception will have two different associated vector:

- One for the exception arising in Non-secure state
- 5 - One for the exception arising in Secure state

This may be useful to reduce the exception latency, because the monitor kernel does not have any more the need to detect the originating state where the exception occurred.

- Implement the Monitor as a new "state", i.e. being able to be in Monitor state (to have the appropriate access rights) and in IRQ (or any other) mode, to see the IRQ (or any other) private registers.

5 **Boot mechanism**

The boot mechanism must respect the following features:

- Keep compatibility with legacy OSes.
- Boot in most privileged mode to ensure the security of the system.

10 As a consequence, Carbon cores will boot in Secure Supervisor mode.

The different systems will then be:

- For systems wanting to run legacy OSes, the S bit is not taken into account and the core will just see it boots in Supervisor mode.
- 15 - For systems wanting to use the Carbon features, the core boots in Secure privileged mode which should be able to configure all secure protections in the system (potentially after swapping to Monitor mode)

Basic Scenario (See Figure 20)

- 20 1. Thread 1 is running in non-secure world (S bit = 0)

This thread needs to perform a secure function => SMI instruction.

2. The SMI instruction makes the core enter the Monitor mode through a dedicated non-secure SMI vector.

25 LR_mon and SPSR_mon are used to save the PC and CPSR of the non secure mode.
The S bit remains unchanged (i.e. non-secure state).

The monitor kernel saves the non-secure context on the monitor.

It also pushes LR_mon and SPSR_mon.

The monitor kernel then changes the "S" bit by writing into the CP15 register.

30 It must keep track that a "secure thread 1" will be started in the secure world (e.g.
by updating a Thread ID table).

for determining the physical address corresponding to that virtual address, and for resolving access permission rights and determining region attributes.

5 The memory system of the data processing apparatus consists of secure memory and non-secure memory, the secure memory being used to store secure data that is intended only to be accessible by the core 10, or one or more other master devices, when that core or other device is operating in a secure mode of operation, and is accordingly operating in the secure domain.

10 In the embodiment of the present invention illustrated in Figure 21, the policing of attempts to access secure data in secure memory by applications running on the core 10 in non-secure mode is performed by the partition checker 222 within the MPU 220, the MPU 220 being managed by the secure operating system, also referred to herein as the secure kernel.

15 In accordance with preferred embodiments of the present invention a non-secure page table 58 is provided within non-secure memory, for example within a non-secure memory portion of external memory 56, and is used to store for each of a number of non-secure memory regions defined within that page table a corresponding
20 descriptor. The descriptor contains information from which the MMU 200 can derive access control information required to enable the MMU to perform the predetermined access control functions, and accordingly in the embodiment described with reference to Figure 21 will provide information about the virtual to physical address mapping, the access permission rights, and any region attributes.

25 Furthermore, in accordance with the preferred embodiments of the present invention, at least one secure page table 58 is provided within secure memory of the memory system, for example within a secure part of external memory 56, which again for a number of memory regions defined within the table provides an associated
30 descriptor. When the processor is operating in a non-secure mode, the non-secure page table will be referenced in order to obtain relevant descriptors for use in

descriptors in the main TLB 208 that are relevant to the particular mode of operation. Hence, if the core 10 is operating in non-secure mode, only those descriptors within the main TLB 208 which have been obtained from the non-secure page table will be checked, whereas if the core 10 is operating in secure mode, only the descriptors
5 within the main TLB that have been obtained from the secure page table will be checked.

If there is a hit within the main TLB as a result of that checking process, then the access control information is extracted from the relevant descriptor and passed
10 back over path 242. In particular, the virtual address portion and the corresponding physical address portion of the descriptor will be routed over path 242 to the micro-TLB 206, for storage in an entry of the micro-TLB, the access permission rights will be loaded into the access permission logic 202, and the region attributes will be loaded into the region attribute logic 204. The access permission logic 202 and region
15 attribute logic 204 may be separate to the micro-TLB, or may be incorporated within the micro-TLB.

At this point, the MMU 200 is then able to process the memory access request since there will now be a hit within the micro-TLB 206. Accordingly, the micro-TLB
20 206 will generate the physical address, which can then be output over path 238 onto the system bus 40 for routing to the relevant memory, this being either on-chip memory such as the TCM 36, cache 38, etc, or one of the external memory units accessible via the external bus interface 42. At the same time, the access permission logic 202 will determine whether the memory access is allowed, and will issue an
25 abort signal back to the core 10 over path 230 if it determines that the core is not allowed to access the specified memory location in its current mode of operation. For example, certain portions of memory, whether in secure memory or non-secure memory, may be specified as only being accessible by the core when that core is operating in supervisor mode, and accordingly if the core 10 is seeking to access such
30 a memory location when in, for example, user mode, the access permission logic 202 will detect that the core 10 does not currently have the appropriate access rights, and

As mentioned earlier, in preferred embodiments, the main TLB 208 can store descriptors from both the secure page table and the non-secure page table, but the memory access requests are only processed by the MMU 200 once the relevant information is stored within the micro-TLB 206. In preferred embodiments, the
5 transfer of information between the main TLB 208 and the micro-TLB 206 is monitored by the partition checker 222 located within the MPU 220 to ensure that, in the event that the core 10 is operating in a non-secure mode, no access control information is transferred into the micro-TLB 206 from descriptors in the main TLB 208 if that would cause a physical address to be generated which is within secure
10 memory.

The memory protection unit is managed by the secure operating system, which is able to set within registers of the CP15 34 partitioning information defining the partitions between the secure memory and the non-secure memory. The partition
15 checker 222 is then able to reference that partitioning information in order to determine whether access control information is being transferred to the micro-TLB 206 which would allow access by the core 10 in a non-secure mode to secure memory. More particularly, in preferred embodiments, when the core 10 is operating in a non-secure mode of operation, as indicated by the domain bit set by the monitor mode
20 within the CP15 domain status register, the partition checker 222 is operable to monitor via path 244 any physical address portion seeking to be retrieved into the micro-TLB 206 from the main TLB 208 and to determine whether the physical address that would then be produced for the virtual address based on that physical address portion would be within the secure memory. In such circumstances, the
25 partition checker 222 will issue an abort signal over path 230 to the core 10 to prevent the memory access from taking place.

It will be appreciated that in addition the partition checker 222 can be arranged to actually prevent that physical address portion from being stored in the micro-TLB
30 206 or alternatively the physical address portion may still be stored within the micro-TLB 206, but part of the abort process would be to remove that incorrect physical

circumstances. Similarly, the region attribute logic 226 will generate cacheable and bufferable signals in the same way that the region attribute logic 204 of the MMU would have generated such signals for memory access requests specified with virtual addresses. Assuming the access is allowed, the access request will then proceed over
5 path 240 onto the system bus 40, from where it is routed to the appropriate memory unit.

For a non-secure access where the access request specifies a physical address, the access request will be routed via path 236 into the partition checker 222, which
10 will perform partition checking with reference to the partitioning information in the CP15 registers 34 in order to determine whether the physical address specifies a location within secure memory, in which event the abort signal will again be generated over path 230.

15 The above described processing of the memory management logic will now be described in more detail with reference to the flow diagrams of Figures 23 and 24. Figure 23 illustrates the situation in which the program running on the core 10 generates a virtual address, as indicated by step 300. The relevant domain bit within the CP15 domain status register 34 as set by the monitor mode will indicate whether
20 the core is currently running in a secure domain or the non-secure domain. In the event that the core is running in the secure domain, the process branches to step 302, where a lookup is performed within the micro-TLB 206 to see if the relevant portion of the virtual address matches with one of the virtual address portions within the micro-TLB. In the event of a hit at step 302, the process branches directly to step 312,
25 where the access permission logic 202 performs the necessary access permission analysis. At step 314, it is then determined whether there is an access permission violation, and if there is the process proceeds to step 316, where the access permission logic 202 issues an abort over path 230. Otherwise, in the absence of such an access permission violation, the process proceeds from step 314 to step 318, where the
30 memory access proceeds. In particular the region attribute logic 204 will output the

violation, in which event an access permission fault abort is issued at step 316. Otherwise, the process proceeds to step 318 where the remainder of the memory access is performed, as discussed earlier.

5 In the event that at step 320 no hit was located in the micro-TLB, the process proceeds to step 322, where a lookup process is performed in the main TLB 208 to determine whether the relevant non-secure descriptor is present. If not, a page table walk process is performed at step 324 by the translation table walk logic 210 in order to retrieve into the main TLB 208 the necessary non-secure descriptor from the non-secure page table. The process then proceeds to step 326, or proceeds directly to step 10 326 from step 322 in the event that a hit within the main TLB 208 occurred at step 322. At step 326, it is determined that the main TLB now contains the valid tagged non-secure descriptor for the virtual address in question, and then at step 328 the partition checker 222 checks that the physical address that would be generated from 15 the virtual address of the memory access request (given the physical address portion within the descriptor) will point to a location in non-secure memory. If not, i.e. if the physical address points to a location in secure memory, then at step 330 it is determined that there is a security violation, and the process proceeds to step 332 where a secure/non-secure fault abort is issued by the partition checker 222.

20

If however the partition checker logic 222 determines that there is no security violation, the process proceeds to step 334, where the micro-TLB is loaded with the sub-section of the relevant descriptor that contains the physical address portion, whereafter at step 336 the memory access is then processed in the earlier described 25 manner.

The handling of memory access requests that directly issue a physical address will now be described with reference to Figure 24. As mentioned earlier, in this scenario, the MMU 200 will be deactivated, this preferably being achieved by the 30 setting within a relevant register of the CP15 registers an MMU enable bit, this setting process being performed by the monitor mode. Hence, at step 350 the core 10 will

It will be appreciated that various other options are also possible. For example, assuming memory access requests may be issued by both secure and non-secure modes specifying virtual addresses, two MMUs could be provided, one for secure access requests and one for non-secure access requests, i.e. MPU 220 in Figure 21 could be replaced by a complete MMU. In such cases, the use of flags with the main TLB of each MMU to define whether descriptors are secure or non-secure would not be needed, as one MMU would store non-secure descriptors in its main TLB, and the other MMU would store secure descriptors in its main TLB. Of course, the partition checker would still be required to check whether an access to secure memory is being attempted whilst the core is in the non-secure domain.

If, alternatively, all memory access requests directly specified physical addresses, an alternative implementation might be to use two MPUs, one for secure access requests and one for non-secure access requests. The MPU used for non-secure access requests would have its access requests policed by a partition checker to ensure accesses to secure memory are not allowed in non-secure modes.

As a further feature which may be provided with either the Figure 21 or the Figure 22 arrangement, the partition checker 222 could be arranged to perform some partition checking in order to police the activities of the translation table walk logic 210. In particular, if the core is currently operating in the non-secure domain, then the partition checker 222 could be arranged to check, whenever the translation table walk logic 210 is seeking to access a page table, that it is accessing the non-secure page table rather than the secure page table. If a violation is detected, an abort signal would preferably be generated. Since the translation table walk logic 210 typically performs the page table lookup by combining a page table base address with certain bits of the virtual address issued by the memory access request, this partition checking may involve, for example, checking that the translation table walk logic 210 is using a base address of a non-secure page table rather than a base address of a secure page table.

external memory 56 of Figure 1, includes within it a non-secure page table 395 specified in a CP15 register 34 by reference to a base address 397. Similarly, within secure memory 400, which again may be within the external memory 56 of Figure 1, a corresponding secure page table 405 is provided which is specified within a duplicate
5 CP15 register 34 by a secure page table base address 407. Each descriptor within the non-secure page table 395 will point to a corresponding non-secure page in non-secure memory 390, whereas each descriptor within the secure page table 405 will define a corresponding secure page in the secure memory 400. In addition, as will be described in more detail later, it is possible for certain areas of memory to be shared
10 memory regions 410, which are accessible by both non-secure modes and secure modes.

Figure 27 illustrates in more detail the lookup process performed within the main TLB 208 in accordance with preferred embodiments. As mentioned earlier, the
15 main TLB 208 includes a domain flag 425 which identifies whether the corresponding descriptor 435 is from the secure page table or the non-secure page table. This ensures that when a lookup process is performed, only the descriptors relevant to the particular domain in which the core 10 is operating will be checked. Figure 27 illustrates an example where the core is running in the secure domain, also referred to
20 as the secure world. As can be seen from Figure 27, when a main TLB 208 lookup is performed, this will result in the descriptors 440 being ignored, and only the descriptors 445 being identified as candidates for the lookup process.

In accordance with preferred embodiments, an additional process ID flag 430,
25 also referred to herein as the ASID flag, is provided to identify descriptors from process specific page tables. Accordingly, processes P1, P2 and P3 may each have corresponding page tables provided within the memory, and further may have different page tables for non-secure operation and secure operation. Further, it will be appreciated that the processes P1, P2, P3 in the secure domain may be entirely
30 separate processes to the processes P1, P2, P3 in the non-secure domain.

In preferred embodiments, the choice between this configuration of the main TLB, and the earlier described configuration with separate secure and non-secure descriptors, can be set by a particular bit provided within the CP15 control registers. In preferred embodiments, this bit would only be set by the secure kernel.

5

In embodiments where the secure application were directly allowed to use a non-secure virtual address, it would be possible to make a non-secure stack pointer available from the secure domain. This can be done by copying a non-secure register value identifying the non-secure stack pointer into a dedicated register within the CP15 registers 34. This will then enable the non-secure application to pass parameters via the stack according to a scheme understood by the secure application.

As described earlier, the memory may be partitioned into non-secure and secure parts, and this partitioning is controlled by the secure kernel using the CP15 registers 34 dedicated to the partition checker 222. The basic partitioning approach is based on region access permissions as definable in typical MPU devices. Accordingly, the memory is divided into regions, and each region is preferably defined with its base address, size, memory attributes and access permissions. Further, when overlapping regions are programmed, the attributes of the upper region take highest priority. Additionally, in accordance with preferred embodiments of the present invention, a new region attribute is provided to define whether that corresponding region is in secure memory or in non-secure memory. This new region attribute is used by the secure kernel to define the part of the memory that is to be protected as secure memory.

25

At the boot stage, a first partition is performed as illustrated in Figure 28. This initial partition will determine the amount of memory 460 allocated to the non-secure world, non-secure operating system and non-secure applications. This amount corresponds to the non-secure region defined in the partition. This information will then be used by the non-secure operating system for its memory management. The rest of the memory 462, 464, which is defined as secure, is unknown by the non-

30

checker 222, and accordingly no access to secure data can be performed in non-secure mode.

5 However, one problem that could occur would be for an application in the non-secure domain to be able to use the cache operations register to invalidate, clean, or flush the cache. It needs to be ensured that such operations could not affect the security of the system. For example, if the non-secure operating system were to invalidate the cache 38 without cleaning it, any secure dirty data must be written to the external memory before being replaced. Preferably, secure data is tagged in the cache,
10 and accordingly can be dealt with differently if desired.

In preferred embodiments, if an "invalidate line by address" operation is executed by a non-secure program, the physical address is checked by the partition checker 222, and if the cache line is a secure cache line, the operation becomes a
15 "clean and invalidate" operation, thereby ensuring that the security of the system is maintained. Further, in preferred embodiments, all "invalidate line by index" operations that are executed by a non-secure program become "clean and invalidate by index" operations. Similarly, all "invalidate all" operations executed by a non-secure program become "clean and invalidate all" operations.

20

Furthermore, with reference to Figure 1, any access to the TCM 36 by the DMA 32 is controlled by the micro-TLB 206. Hence, when the DMA 32 performs a lookup in the TLB to translate its virtual address into a physical one, the earlier described flags that were added in the main TLB allow the required security checking
25 to be performed, just as if the access request had been issued by the core 10. Further, as will be discussed later, a replica partition checker is coupled to the external bus 70, preferably being located within the arbiter/decoder block 54, such that if the DMA 32 directly accesses the memory coupled to the external bus 70 via the external bus interface 42, the replica partition checker connected to that external bus checks the
30 validity of the access. Furthermore, in certain preferred embodiments, it would be possible to add a bit to the CP15 registers 34 to define whether the DMA controller 32

By default, it is envisaged that the TCM would be used only by non-secure operating systems, as in this scenario the non-secure operating system would not need
5 to be changed.

As mentioned earlier, in addition to the provision of the partition checker 222 within the MPU 220, preferred embodiments of the present invention also provide an analogous partition checking block coupled to the external bus 70, this additional
10 partition checker being used to police accesses to memory by other master devices, for example the digital signal processor (DSP) 50, the DMA controller 52 coupled directly to the external bus, the DMA controller 32 connectable to the external bus via the external bus interface 42, etc. As mentioned earlier, the entire memory system can consist of several memory units, and a variety of these may exist on the external bus
15 70, for example the external memory 56, boot ROM 44, or indeed buffers or registers 48, 62, 66 within peripheral devices such as the screen driver 46, I/O interface 60, key storage unit 64, etc. Furthermore, different parts of the memory system may need to be defined as secure memory, for example it may be desired that the key buffer 66 within the key storage unit 64 should be treated as secure memory. If an access to
20 such secure memory were to be attempted by a device coupled to the external bus, then it is clear that the earlier described memory management logic 30 provided within the chip containing the core 10 would not be able to police such accesses.

Figure 31 illustrates how the additional partition checker 492 coupled to the
25 external bus, also referred to herein as the device bus, is used. The external bus would typically be arranged such that whenever memory access requests were issued onto that external bus by devices, such as devices 470, 472, those memory access requests would also include certain signals on the external bus defining the mode of operation, for example privileged, user, etc. In accordance with preferred embodiments of the
30 present invention the memory access request also involves issuance of a domain signal onto the external bus to identify whether the device is operating in secure mode

device 470 were to assert a memory access request in non-secure mode to a register 482 within the screen driver 480 that was marked as secure memory, then the screen driver 480 would determine that the S bit was not asserted, and would not process the memory access request. Accordingly, it is envisaged that with appropriate design of
5 the various memory devices, it may be possible to avoid the need for a partition checker 492 to be provided separately on the external bus.

Figure 33 shows different modes and applications running on a processor. The dashed lines indicate how different modes and/or applications can be separated and
10 isolated from one another during monitoring of the processor according to an embodiment of the present invention.

The ability to monitor a processor to locate possible faults and discover why an application is not performing as expected is extremely useful and many processors
15 provide such functions. The monitoring can be performed in a variety of ways including debug and trace functions.

In the processor according to the present technique debug can operate in several modes including halt debug mode and monitor debug mode. These modes are
20 intrusive and cause the program running at the time to be stopped. In halt debug mode, when a breakpoint or watchpoint occurs, the core is stopped and isolated from the rest of the system and the core enters debug state. On entry the core is halted, the pipeline is flushed and no instructions are pre-fetched. The PC is frozen and any interrupts (IRQ and FIQ) are ignored. It is then possible to examine the core internal
25 state (via the JTAG serial interface) as well as the state of the memory system. This state is invasive to program execution, as it is possible to modify current mode, change register contents, etc. Once Debug is terminated, the core exits from the Debug State by scanning in the Restart instruction through the Debug TAP (test access port). Then the program resumes execution.

such as debug and trace which are conventionally allowed access to the whole system are a potential source of data leakage between the domains.

In the example given above of a secure and non-secure domain or world, secure data must not be available to the non-secure world. Furthermore, if debug is permitted, in secure world, it may be advantageous for some of the data within secure world to be restricted or hidden. The hashed lines in Figure 33 shows some examples of possible ways to segment data access and provide different levels of granularity. In Figure 33, monitor mode is shown by block 500 and is the most secure of all the modes and controls switching between secure and non-secure worlds. Below monitor mode 500 there is a supervisor mode, this comprises secure supervisor mode 510 and non-secure supervisor mode 520. Then there is non-secure user mode having applications 522 and 524 and secure user mode with applications 512, 514 and 516. The monitoring modes (debug and trace) can be controlled to only monitor non-secure mode (to the left of hashed line 501). Alternatively the non-secure domain or world and the secure user mode may be allowed to be monitored (left of 501 and the portion right of 501 that lies below 502). In a further embodiment the non-secure world and certain applications running in the secure user domain may be allowed, in this case further segmentation by hashed lines 503 occurs. Such divisions help prevent leakage of secure data between different users who may be running the different applications. In some controlled cases monitoring of the entire system may be allowed. According to the granularity required the following parts of the core need to have their access controlled during monitoring functions.

There are four registers that can be set on a Debug event; the instruction Fault Status Register (IFSR), Data Fault Status Register (DFSR), Fault Address Register (FAR), and Instruction Fault Address Register (IFAR). These registers should be flushed in some embodiments when going from secure world to non-secure world to avoid any leak of data.

function to certain secure portions of the core by only allowing initialisation under certain conditions.

Embodiments of the present technique seek to restrict entry into monitoring
5 functions with the following granularity:

By controlling separately intrusive and observable (trace) debug;

By allowing debug entry in secure user mode only or in the whole secure world;

By allowing debug in secure user mode only and moreover taking account of
10 the thread ID (application running).

In order to control the initiation of a monitoring function it is important to be aware of how the functions can be initiated. Figure 34 shows a table illustrating the possible ways of initiating a monitoring function, the type of monitoring function that
15 is initiated and the way that such an initiation instruction can be programmed.

Generally, these monitoring instructions can be entered via software or via hardware, i.e. via the JTAG controller. In order to control the initiation of monitoring functions, control values are used. These comprise enable bits which are condition
20 dependent and thus, if a particular condition is present, monitoring is only allowed to start if the enable bit is set. These bits are stored on a secure register CP14 (debug and status control register, DSCR), which is located in ICE 530 (see Figure 41).

In a preferred embodiment there are four bits that enable/disable intrusive and
25 observable debug, these comprise a secure debug enable bit, a secure trace enable bit, a secure user-mode enable bit and a secure thread aware enable bit. These control values serve to provide a degree of controllable granularity for the monitoring function and as such can help stop leakage of data from a particular domain. Figure 35 provides a summary of these bits and how they can be accessed.

In functional mode, SE (Scan Enable) is clear and the register cell works as a single register cell. In test or debug mode, SE is set and input data can come from SI input (Scan In) instead of D input.

5 **Scan chain**

All scan chain cells are chained in scan chain as shown in figure 38.

In functional mode, SE is clear and all register cells can be accessed normally and interact with other logic of the circuit. In Test or Debug mode, SE is set and all registers are chained between each other in a scan chain. Data can come from the first scan chain cell and can be shifted through any other scan chain cell, at the cadence of each clock cycle. Data can be shifted out also to see the contents of the registers.

15 **TAP controller**

A debug TAP controller is used to handle several scan chains. The TAP controller can select a particular scan chain: it connects “Scan In” and “Scan Out” signals to that particular scan-chain. Then data can be scanned into the chain, shifted, or scanned out. The TAP controller is controlled externally by a JTAG port interface. Figure 39 schematically illustrates a TAP controller

25 **JTAG Selective Disable Scan Chain Cell**

For security reasons, some registers might not be accessible by scan chain, even in debug or test mode. A new input called JADI (JTAG Access Disable) can allow removal dynamically or statically of a scan chain cell from a whole scan chain, without modifying the scan chain structure in the integrated circuit. Figures 40A and B schematically show this input.

It should be noted that by default debug (intrusive and observable – trace) are only available in non-secure world. To enable them to be available in secure world the control value enable bits need to be set.

5 The advantages of this are that debug can always be initiated by users to run in non-secure world. Thus, although access to secure world is not generally available to users in debug this may not be a problem in many cases because access to this world is limited and secure world has been fully verified at board stage prior to being made available. It is therefore foreseen that in many cases debugging of the secure world
10 will not be necessary. A secure supervisor can still initiate debug via the software route of writing CP14 if necessary.

Figure 42 schematically shows the control of debug initialisation. In this figure a portion of the core 600 comprises a storage element 601 (which may be a
15 CP15 register as previously discussed) in which is stored a secure status bit S indicative of whether the system is in secure world or not. Core 600 also comprises a register 602 comprising bits indicative of the mode that the processor is running in, for example user mode, and a register 603 providing a context identifier that identifies the application or thread that is currently running on the core.

20 When a breakpoint is reached comparator 610, which compares a breakpoint stored on register 611 with the address of the core stored in register 612, sends a signal to control logic 620. Control logic 620 looks at the secure state S, the mode 602 and the thread (context identifier) 603 and compares it with the control values and
25 condition indicators stored on register CP14. If the system is not operating in secure world, then a “enter debug” signal will be output at 630. If however, the system is operating in secure world, the control logic 620 will look at the mode 602, and if it is in user mode will check to see if user mode enable and debug enable bits are set. If they are then debug will be initialised provided that a thread aware bit has not been
30 initialised. The above illustrates the hierarchical nature of the control values.

It is the same for debug granularity concerning observable debug. Figure 43B shows a summary of secure debug granularity in this case, here default values at reset are also represented in grey colour.

5

Note that Secure user-mode debug enable bit and Secure thread-aware debug enable bit are commonly used for intrusive and observable debug.

10 A thread aware initialisation bit is stored in register CP14 and indicates if granularity by application is required. If the thread aware bit has been initialised, the control logic will further check that the application identifier or thread 603 is one indicated in the thread aware control value, if it is, then debug will be initialised. If either of the user mode or debug enable bits are not set or the thread aware bit is set and the application running is not one indicated in the thread aware control value, then
15 the breakpoint will be ignored and the core will continue doing what it was doing and debug will not be initialised.

In addition to controlling initialisation of monitoring functions, the capture of diagnostic data during a monitor function can also be controlled in a similar way. In
20 order to do this the core must continue to consider both the control values, i.e. the enable bits stored in register CP14 and the conditions to which they relate during operation of the monitoring function.

Figure 44 shows schematically granularity of a monitoring function while it is
25 running. In this case region A relates to a region in which it is permissible to capture diagnostic data and region B relates to region in which control values stored in CP14 indicate that it is not possible to capture diagnostic data.

Thus, when debug is running and a program is operating in region A,
30 diagnostic data is output in a step-by-step fashion during debug. When operation switches to Region B, where the capture of diagnostic data is not allowed, debug no

If we initialise debug in halt debug mode all registers are accessible (non-secure and secure register banks) and the whole memory can be dumped, except the bits dedicated to control debug.

5 Debug halt mode can be entered from whatever mode and from whatever domain. Breakpoints and watchpoints can be set in secure or in non-secure memory. In debug state, it is possible to enter secure world by simply changing the S bit via an MCR instruction.

10 As debug mode can be entered when secure exceptions occur, the vector trap register is extended with new bits which are;

SMI vector trapping enable

Secure data abort vector trapping enable

Secure prefetch abort vector trapping enable

15 Secure undefined vector trapping enable.

In monitor debug mode, if we allow debug everywhere, even when an SMI is called in non-secure world, it is possible to enter secure world in step-by-step debug. When a breakpoint occurs in secure domain, the secure abort handler is operable to
20 dump secure register bank and secure memory.

The two abort handlers in secure and in non-secure world give their information to the debugger application so that debugger window (on the associated debug controlling PC) can show the register state in both secure and non-secure worlds.

25

Figure 45A shows what happens when the core is configured in monitor debug mode and debug is enabled in secure world. Figure 45B shows what happens when the core is configured in monitor debug mode and the debug is disabled in secure world. This later process will be described below.

30

Intrusive debug at production stage

secure world, step-by-step is possible but whenever an SMI is executed secure function is executed entirely in other words an XWSI only "step-over" is allowed while "step-in" and "step-over" are possible on all other instructions. XWSI is thus considered an atomic instruction.

5

Once secure debug is disabled, we have the following restrictions:

Before entering monitor mode:

Breakpoints and watchpoints are only taken into account in non-secure world. If bit
10 S is set, breakpoints/watchpoints are bypassed. Note that watchpoints units are also accessible with MCR/MRC (CP14) which is not a security issue as breakpoint/watchpoint has no effect in secure memory.

BKPT are normally used to replace the instruction on which breakpoint is set. This
15 supposes to overwrite this instruction in memory by BKPT instruction, which will be possible only in non-secure mode.

Vector Trap Register concerns non-secure exceptions only. All extended trapping enable bits explained before have no effect. Data abort and Pre-fetch abort enable bits
20 should be disabled to avoid the processor being forced in to an unrecoverable state.

Via JTAG, we have the same restrictions as for halt mode (S bit cannot be modified, etc)

Once in monitor mode (non-secure abort mode)

25 The non-secure abort handler can dump non-secure world and has no visibility on secure banked registers as well as secure memory.

Executes secure functions with atomic SMI instruction

S bit cannot be changed to force secure world entry.

Mode bits can not be changed if debug is permitted in secure supervisor mode only.

30

Note that if an external debug request (EDBGRQ) occurs,

- S_undef bit: should only be set when debug is enable in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- SMI bit: should only be set when debug is enabled in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- 5 • FIQ, IRQ, D_abort, P_abort, SWI, undef bits: correspond to non-secure exceptions, so they are valid even if debug in secure world is disabled. Note that D_abort and P_abort should not be asserted high in monitor mode.
- Reset bit: as we enter secure world when reset occurs, this bit is valid only when debug in secure world is enabled, otherwise it has no effect.

10

Although a particular embodiment of the invention has been described herein, it will be apparent that the invention is not limited thereto, and that many modifications and additions may be made within the scope of the invention. For example, various combinations of the features of the following dependent could be

15 made with the features of the independent claims without departing from the scope of the present invention.

5. A data processing apparatus as claimed in any preceding claim, wherein the partition checking logic is provided within an arbiter coupled to the device bus to arbitrate between memory access requests issued on the device bus.

5

6. A data processing apparatus as claimed in any preceding claim, wherein in said at least one non-secure mode the device is operable under the control of a non-secure operating system, and in said at least one secure mode the device is operable under the control of a secure operating system.

10

7. A data processing apparatus as claimed in any preceding claim, wherein the device is a chip incorporating a processor, the chip further comprising a memory management unit operable, when the processor generates the memory access request, to perform one or more predetermined access control functions to control issuance of the memory access request onto the device bus.

15

8. A data processing apparatus as claimed in Claim 7, wherein the chip further comprises:

on-chip memory coupled to the processor via a system bus, the on-chip memory operable to store data required by the processor, the on-chip memory comprising secure on-chip memory for storing secure data and non-secure on-chip memory for storing non-secure data; and

20

on-chip partition checking logic coupled to the system bus and operable whenever the memory access request is generated by the processor when operating in said non-secure mode to detect if the memory access request is seeking to access either the secure memory or the secure on-chip memory, and upon such detection to prevent the access specified by that memory access request.

25

9. A method of controlling access to a memory in a data processing apparatus, the data processing apparatus comprising a device bus, a device coupled to the device bus and operable in a plurality of modes and either a secure domain or a non-secure

30

and in said at least one secure mode the device is operable under the control of a secure operating system.

15. A method as claimed in any of claims 9 to 14, wherein the device is a chip
5 incorporating a processor, the chip further comprising a memory management unit, when the processor generates the memory access request, the method comprising the step of:

employing the memory management unit to perform one or more predetermined access control functions to control issuance of the memory access request onto the
10 device bus.

16. A method as claimed in Claim 15, wherein the chip further comprises on-chip memory coupled to the processor via a system bus, the on-chip memory operable to store data required by the processor, the on-chip memory comprising secure on-chip
15 memory for storing secure data and non-secure on-chip memory for storing non-secure data, and on-chip partition checking logic coupled to the system bus, the method further comprising the steps of:

whenever the memory access request is generated by the processor when operating in said non-secure mode, employing the on-chip partition checking logic to
20 detect if the memory access request is seeking to access either the secure memory or the secure on-chip memory; and

upon such detection, preventing the access specified by that memory access request.

1/39

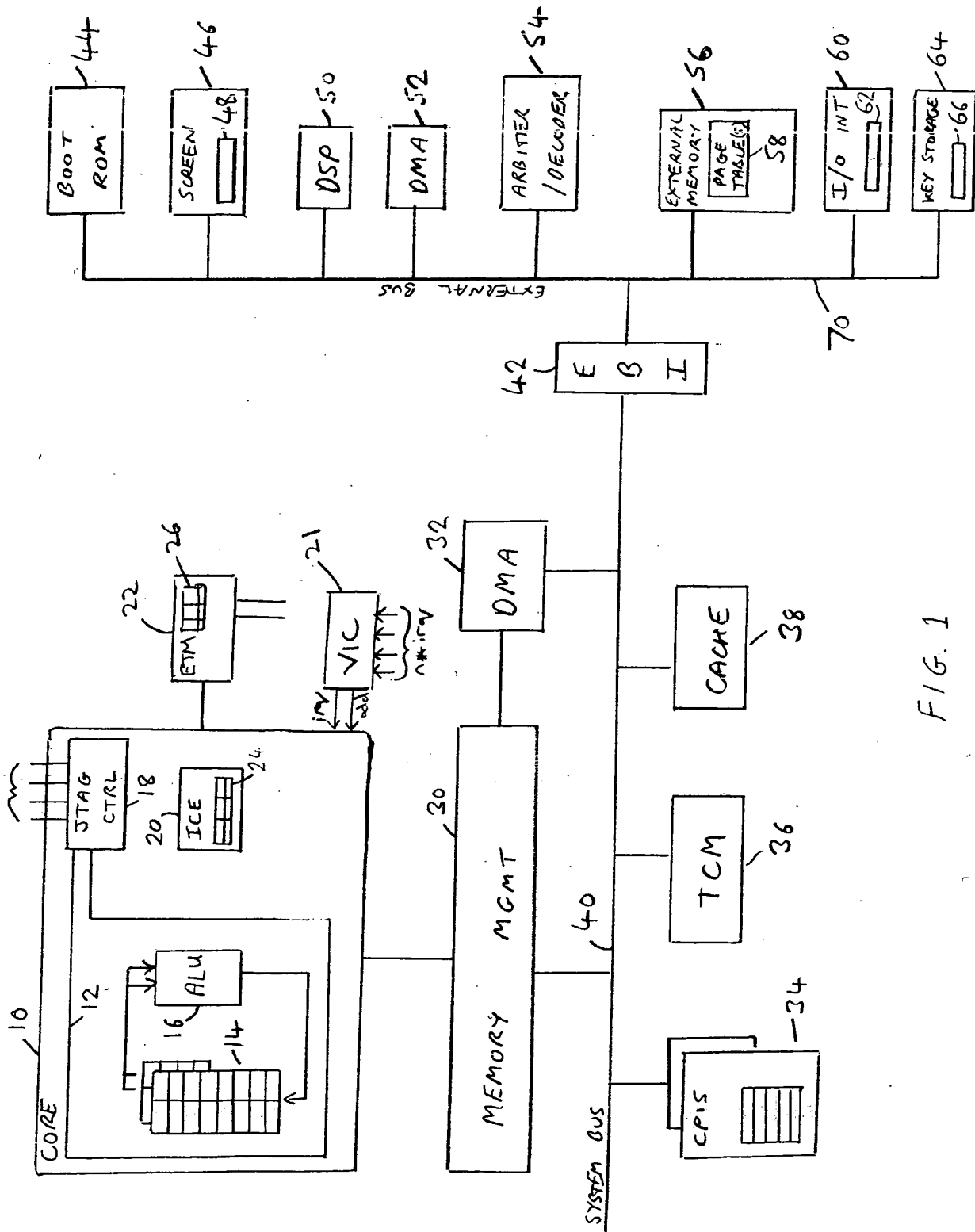


FIG. 1

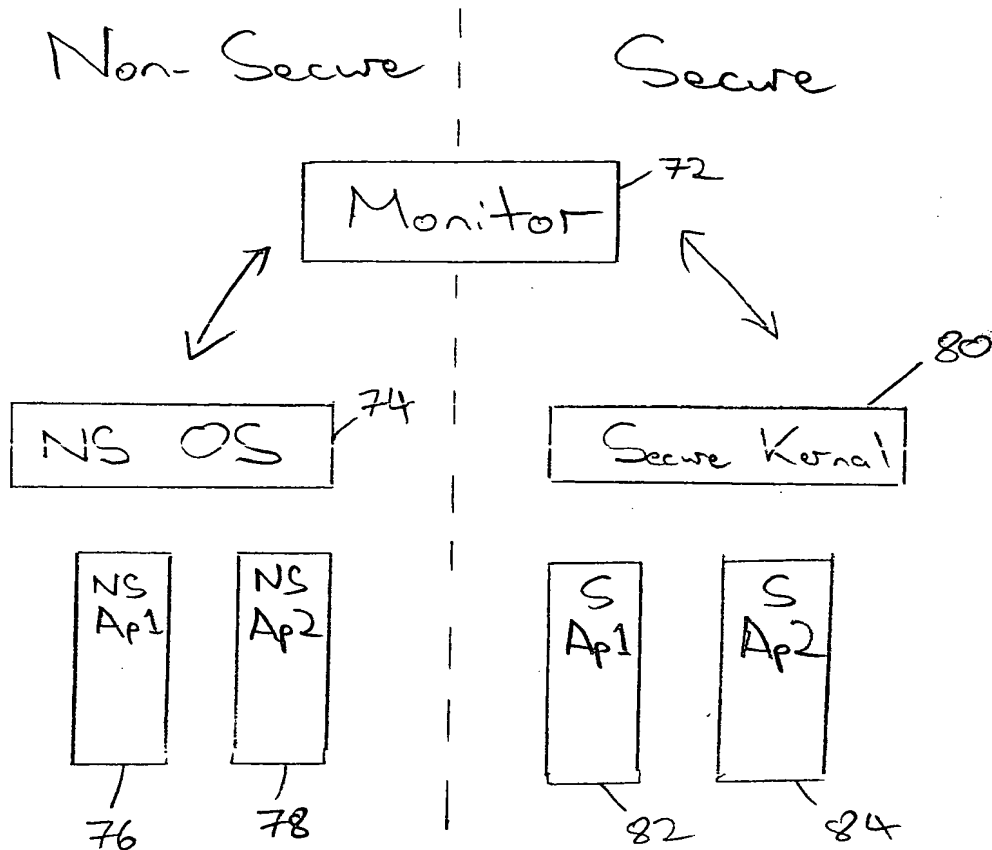


Fig. 2

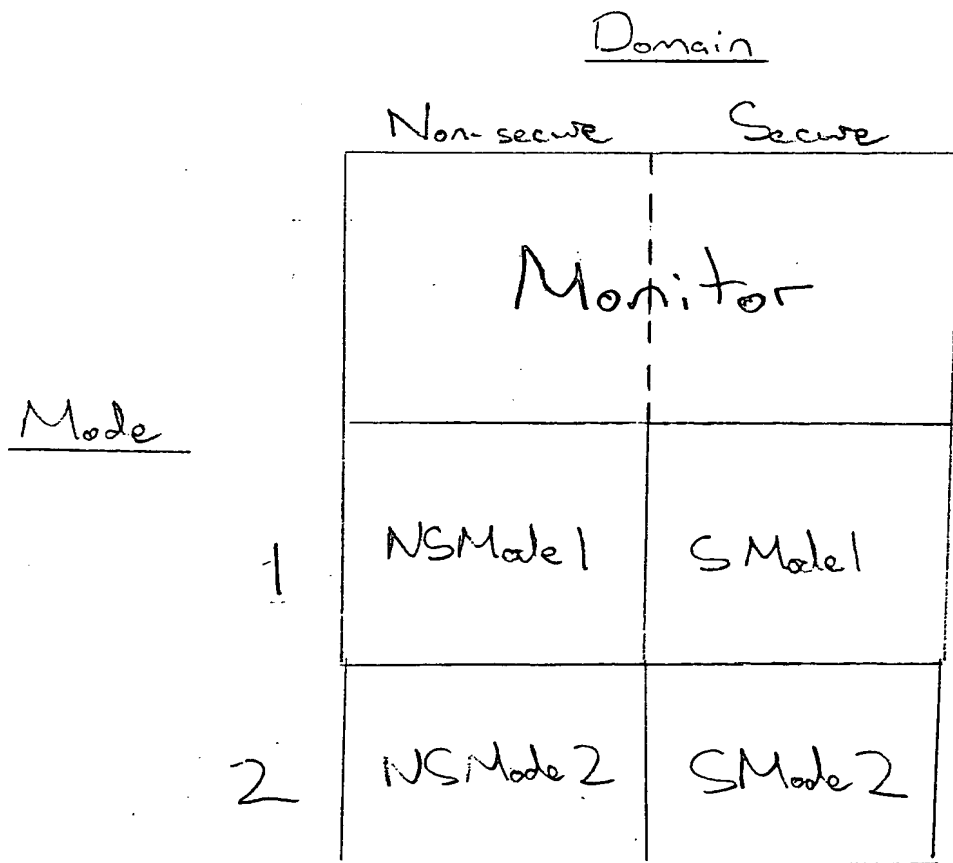


Fig. 3

NS S 3/39

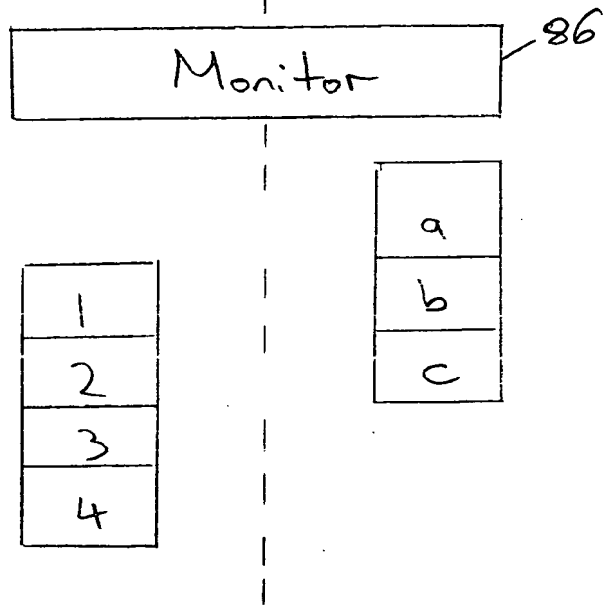


Fig. 4

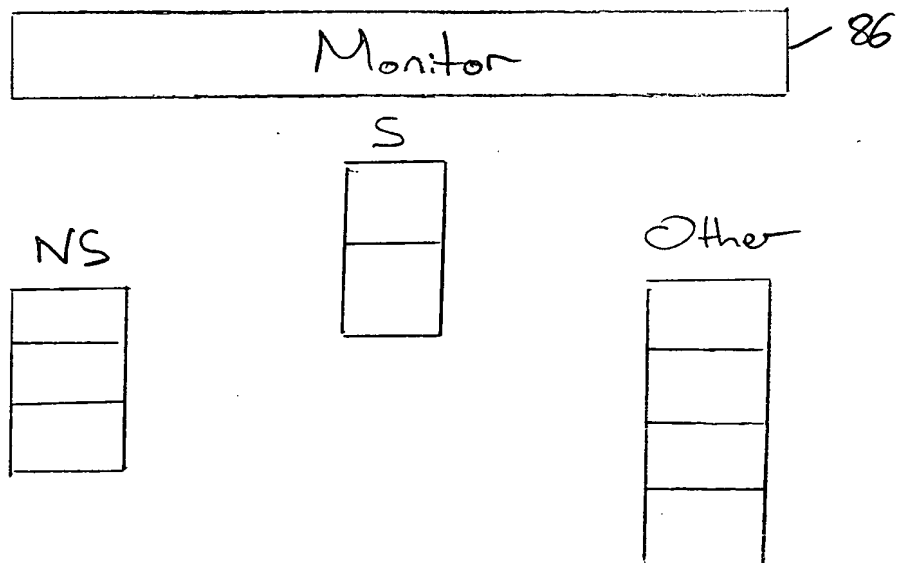
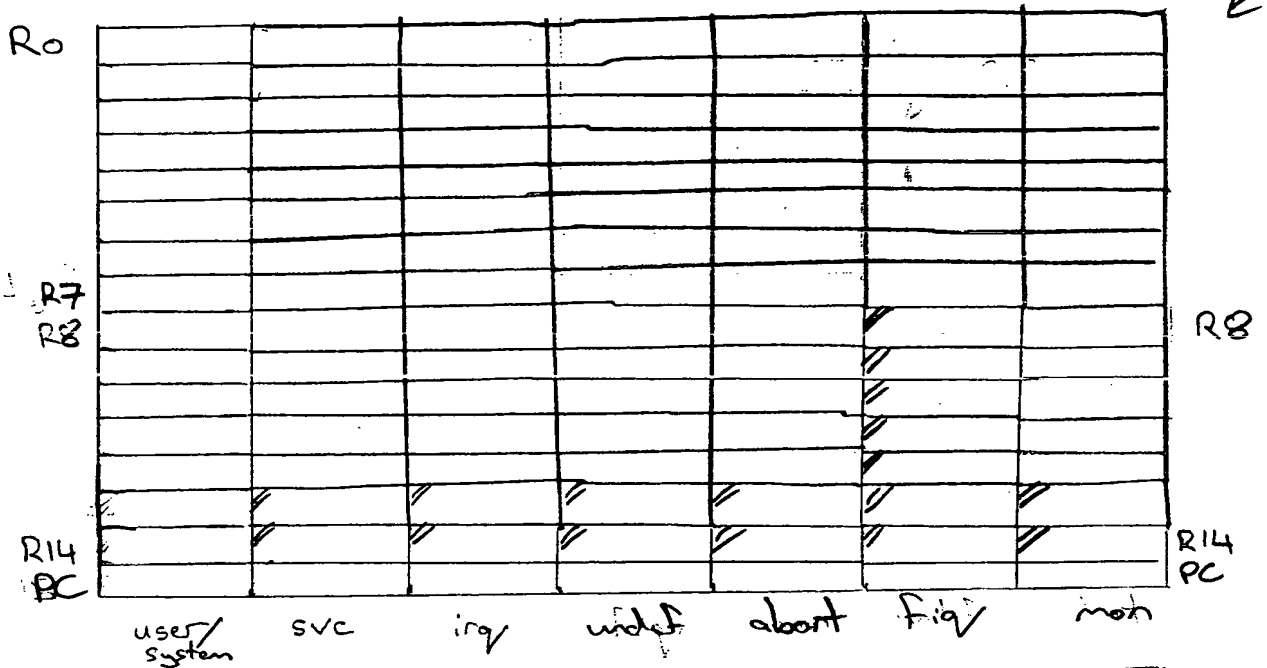


Fig. 5

4/39



CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_irq	SPSR_undef	SPSR_abort	SPSR_fig	SPSR_mon

// = private to mode

Fig. 6

88

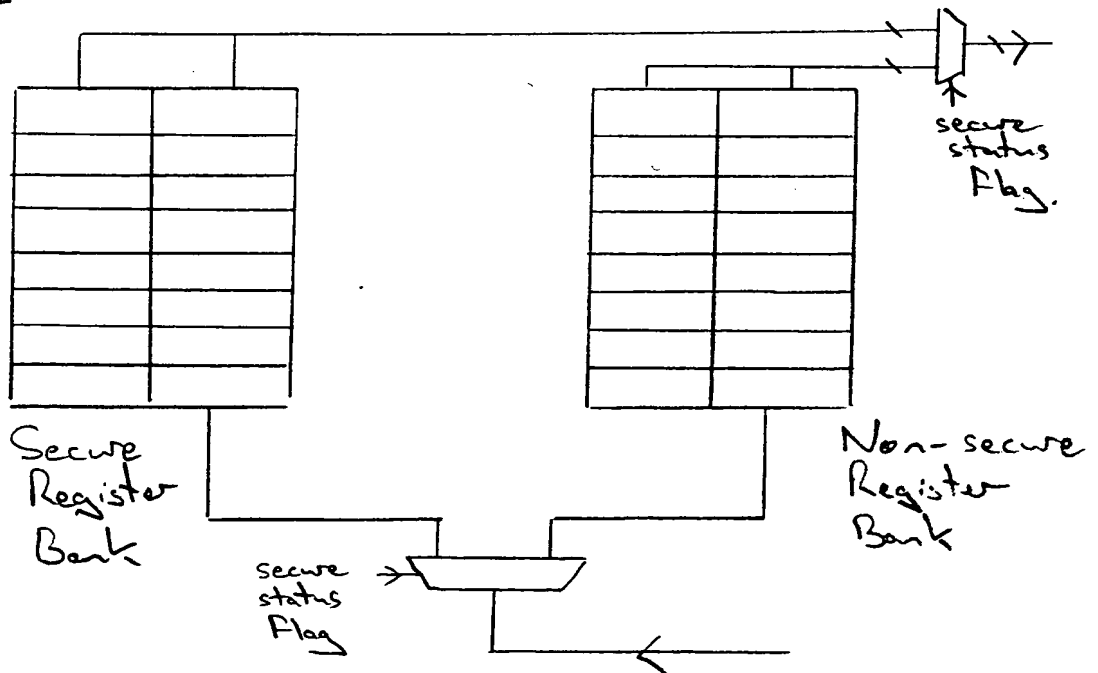


Fig. 7

5/39

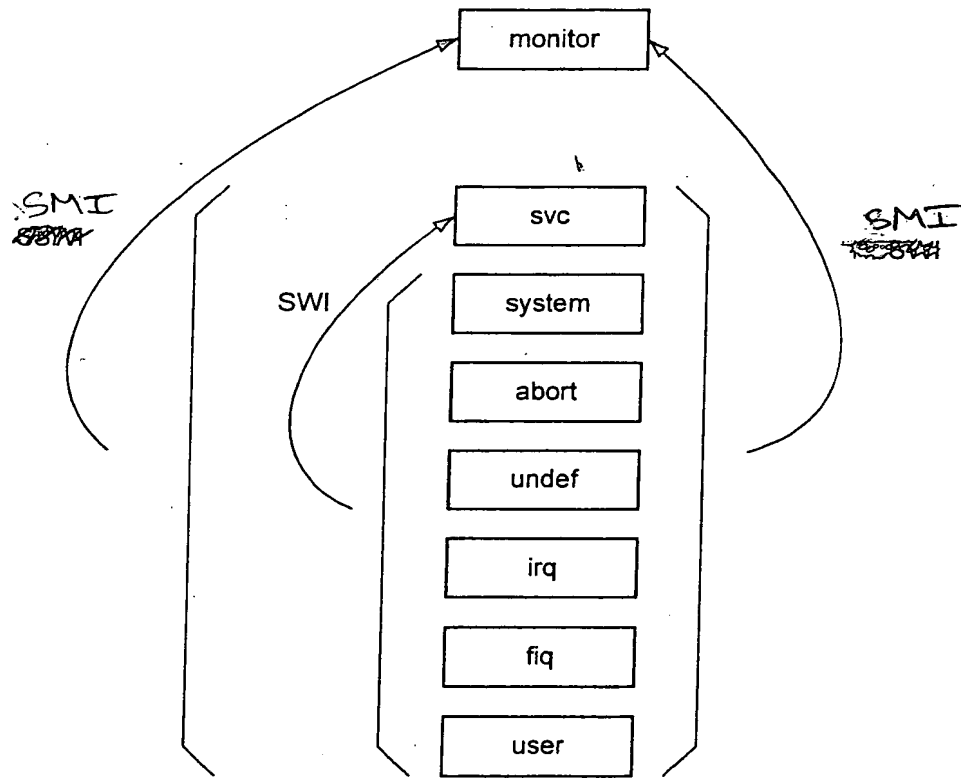


Fig. 8

6/39

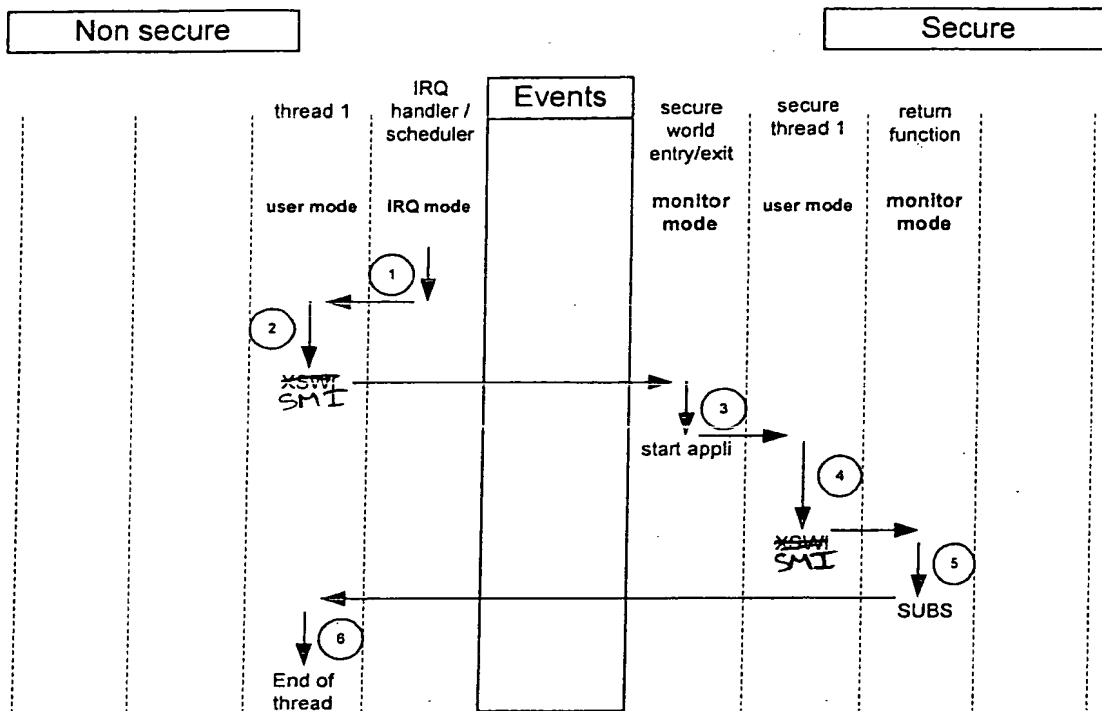


Fig. 9

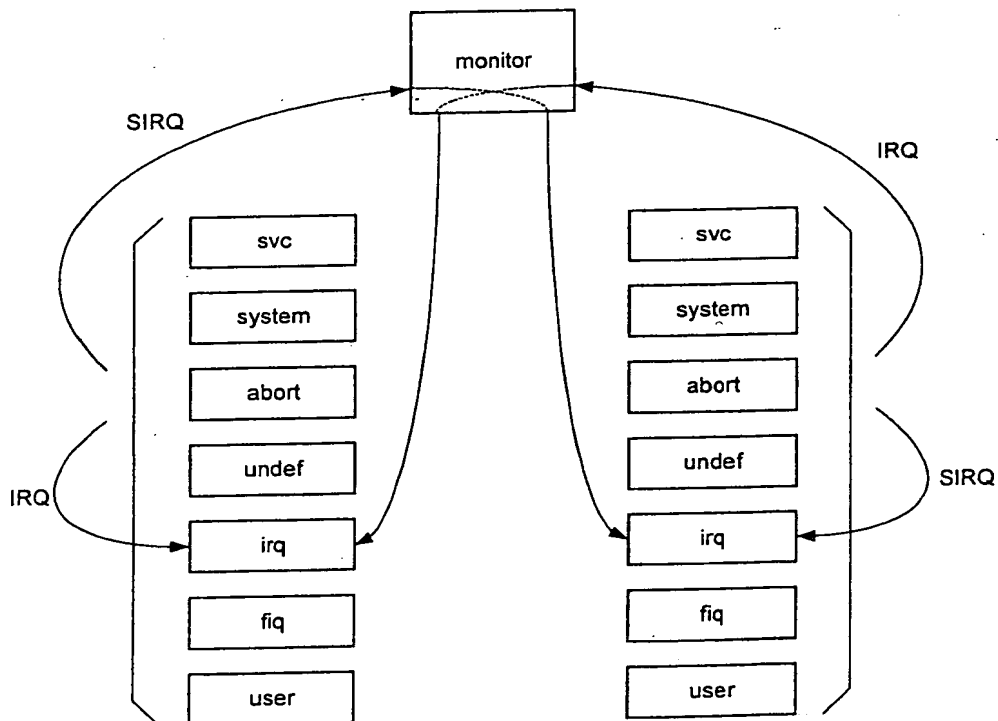


Fig. 10

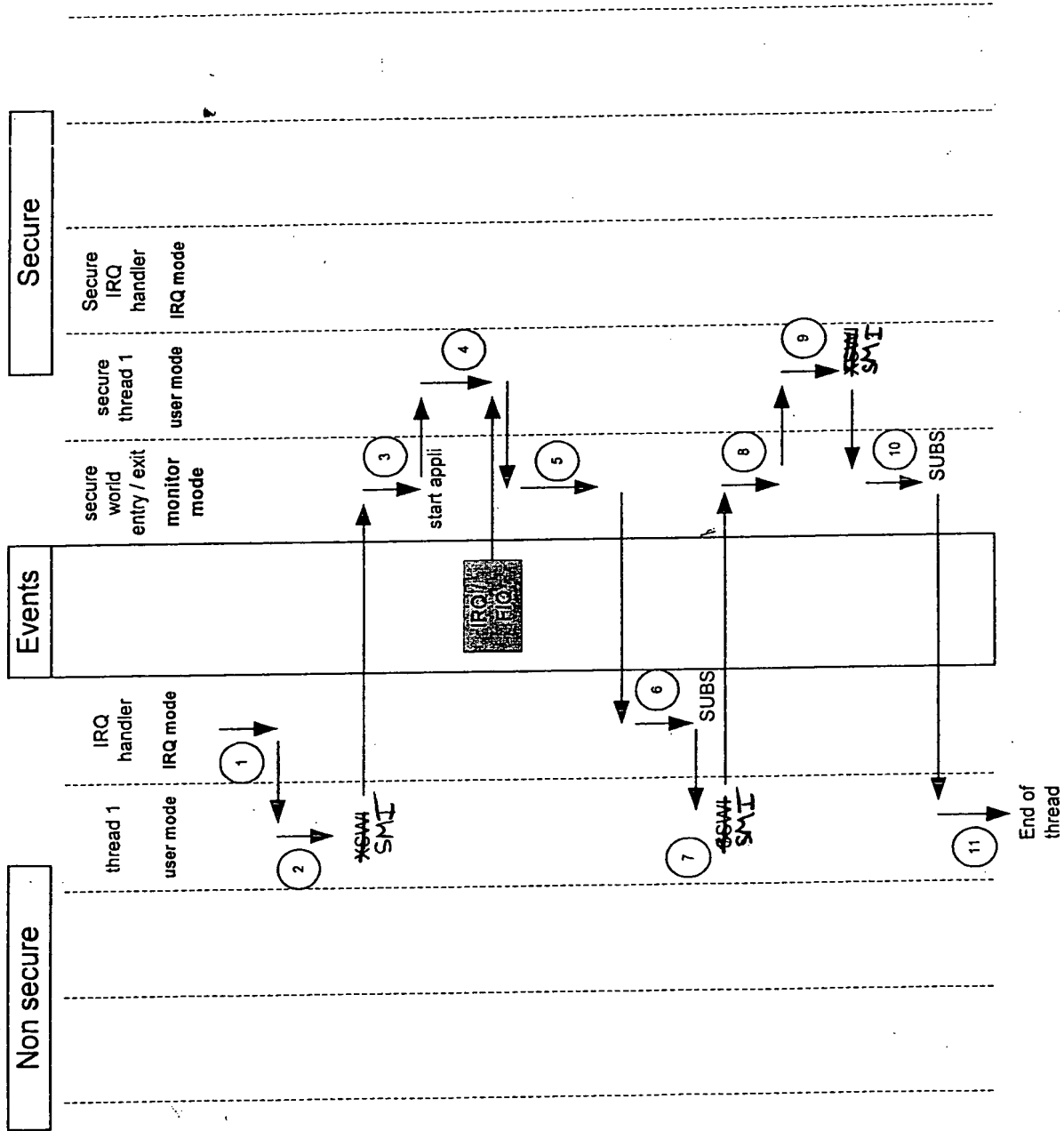


Fig. 11A

8/39

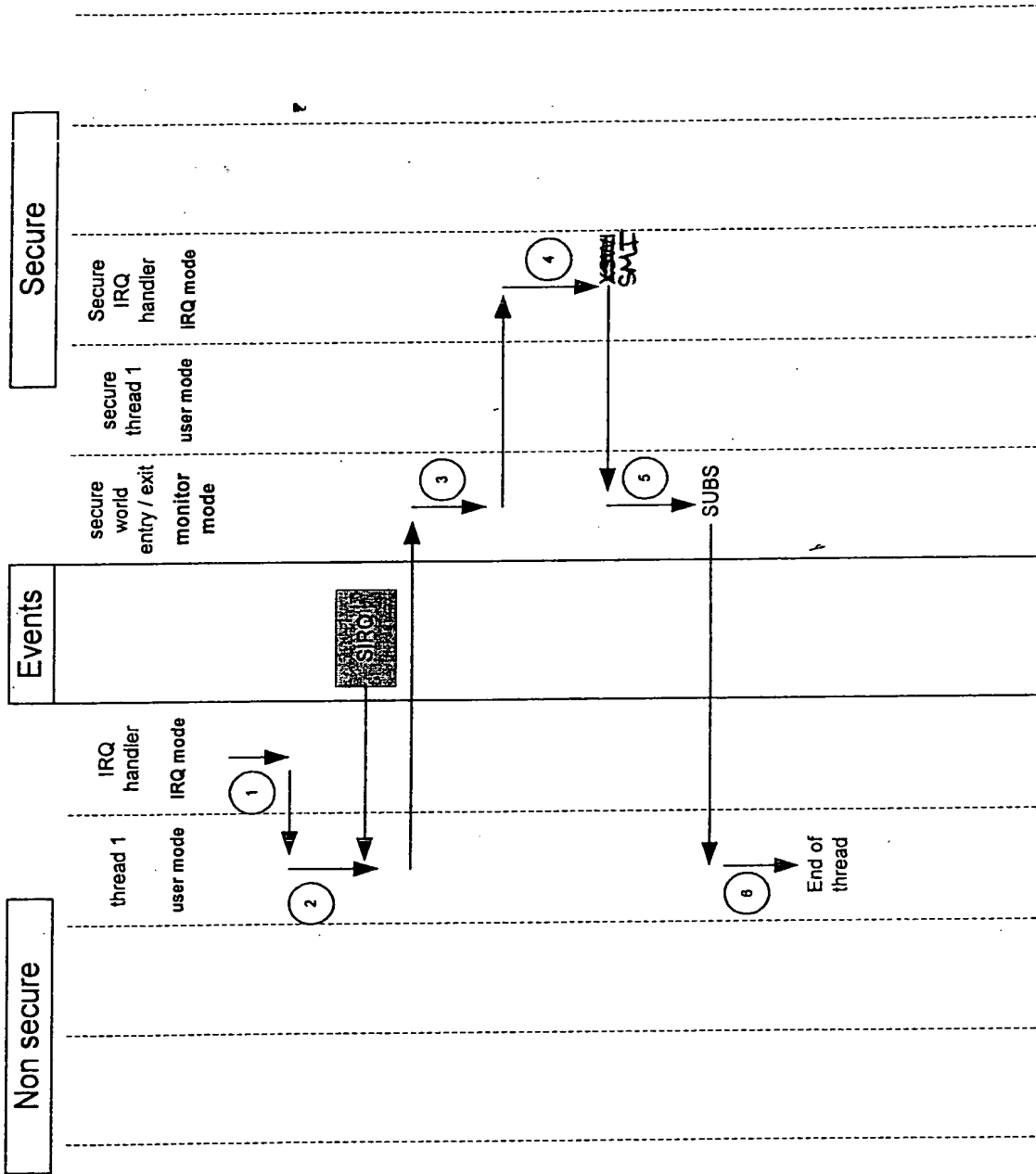
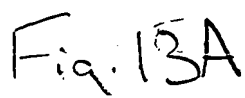
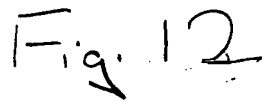


Fig. 11B



10/39

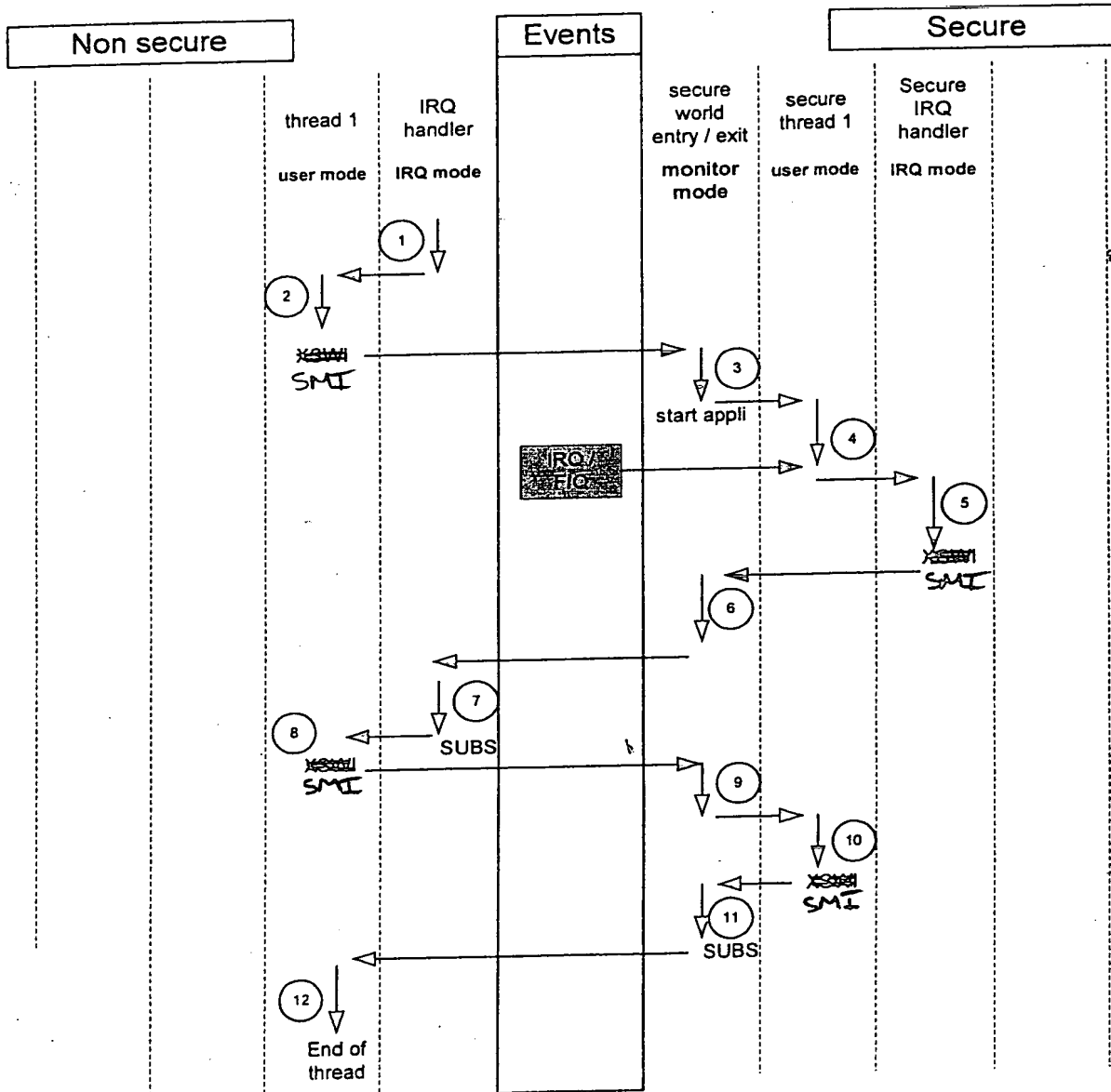


Fig. 13B

11/39

Exception	Vector offset	Corresponding mode
Reset	0x00	Supervisor mode
Under	0x04	Monitor mode / Under mode
SWI	0x08	Supervisor mode / Monitor mode
Prefetch abort	0x0C	Abort mode / Monitor mode
Data abort	0x10	Abort mode / Monitor mode
IRQ / SIRQ	0x18	IRQ mode / Monitor mode
FIQ	0x1C	FIQ mode / Monitor mode
SMT	0x14	Under mode / Monitor mode

Fig. 14

Monitor

Reset	VM0
Under	VM1
SWI	VM2
Prefetch abort	VM3
Data abort	VM4
IRQ / SIRQ	VM5
FIQ	VM6
SMT	VM7

Secure

Reset	VS0
Under	VS1
SWI	VS2
Prefetch abort	VS3
Data abort	VS4
IRQ / SIRQ	VS5
FIQ	VS6
SMT	VS7

Non-Secure

Reset	VNS0
Under	VNS1
SWI	VNS2
Prefetch abort	VNS3
Data abort	VNS4
IRQ / SIRQ	VNS5
FIQ	VNS6
SMT	VNS7

Fig. 15

CPI5 Exception Trap Mask Register (15) only NS world exceptions illustrated

0	1	1	1	1	0	1	
Reset	SMT	SWI	Prefetch Abort	Data Abort	IRQ	SIRQ	FIQ

1 = Mon(S)

0 = NS



OR via hardware / external

pin

value

Fig 16

12/39

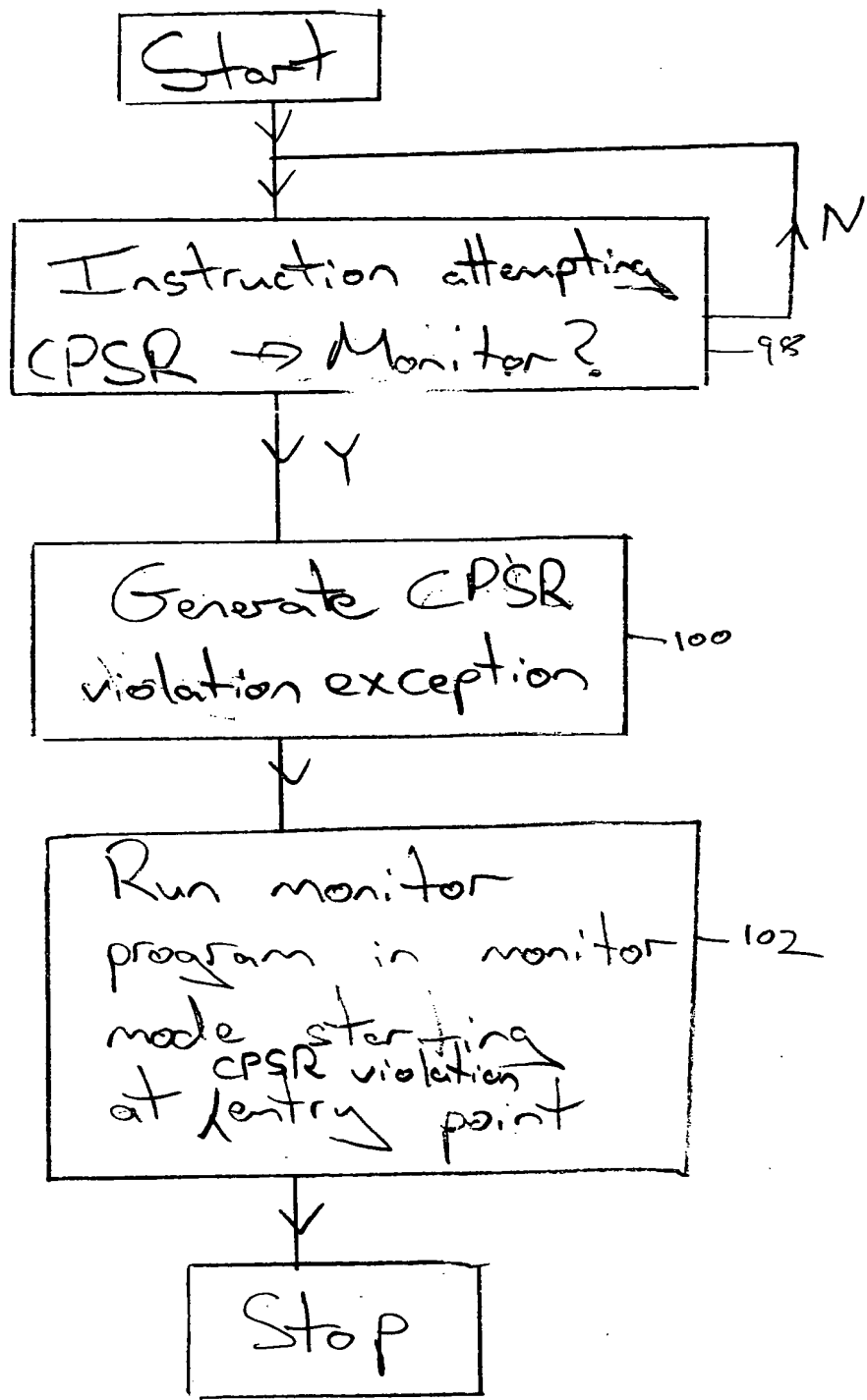


Fig. 17

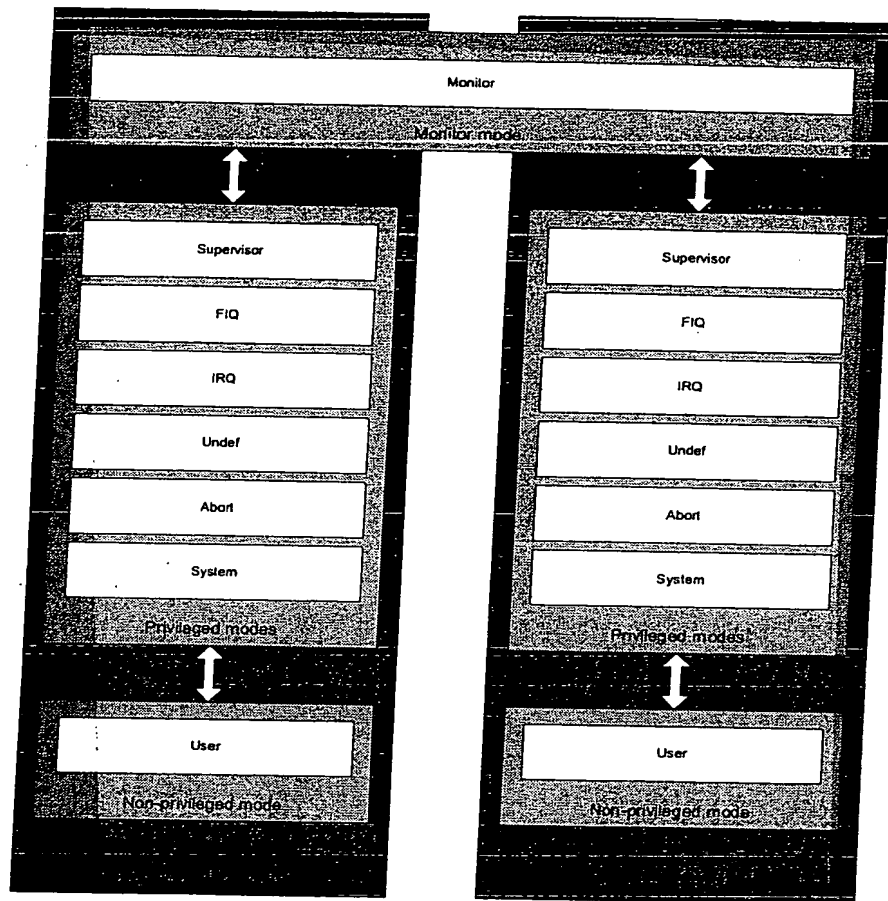


FIGURE 18

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt	Monitor
R0	R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq	R8
R9	R9	R9	R9	R9	R9	R9_fiq	R9
R10	R10	R10	R10	R10	R10	R10_fiq	R10
R11	R11	R11	R11	R11	R11	R11_fiq	R11
R12	R12	R12	R12	R12	R12	R12_fiq	R12
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq	R13_mon
R14	R14	R14_svc	R14_sbt	R14_und	R14_irq	R14_fiq	R14_mon
PC	PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq	SPSR_mon

FIGURE 19

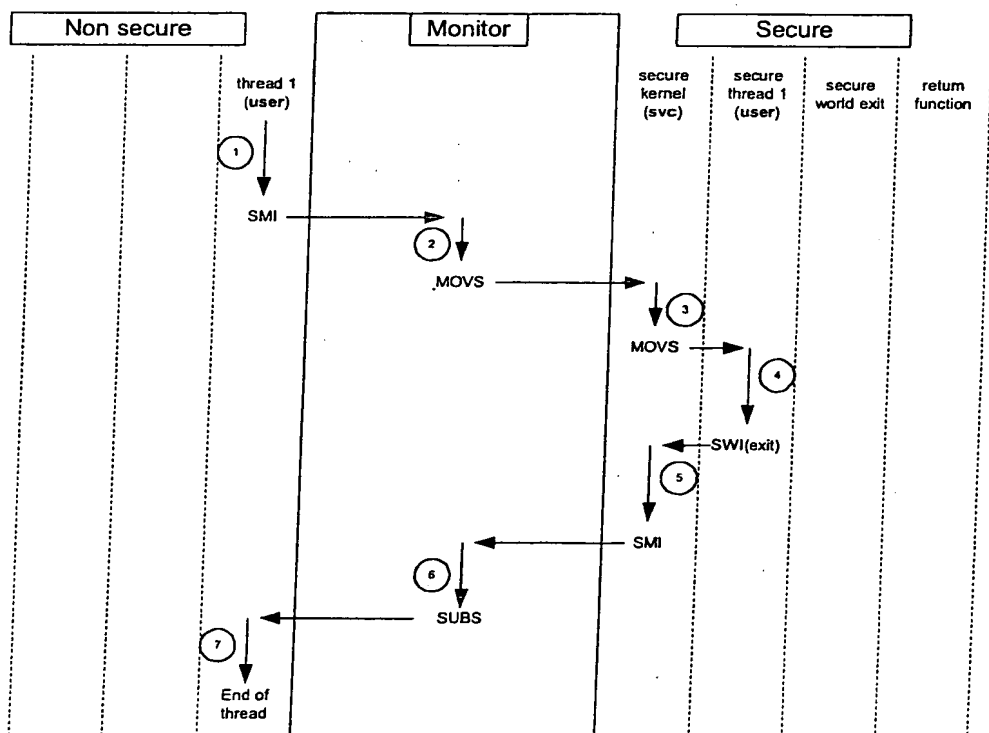


FIGURE 20

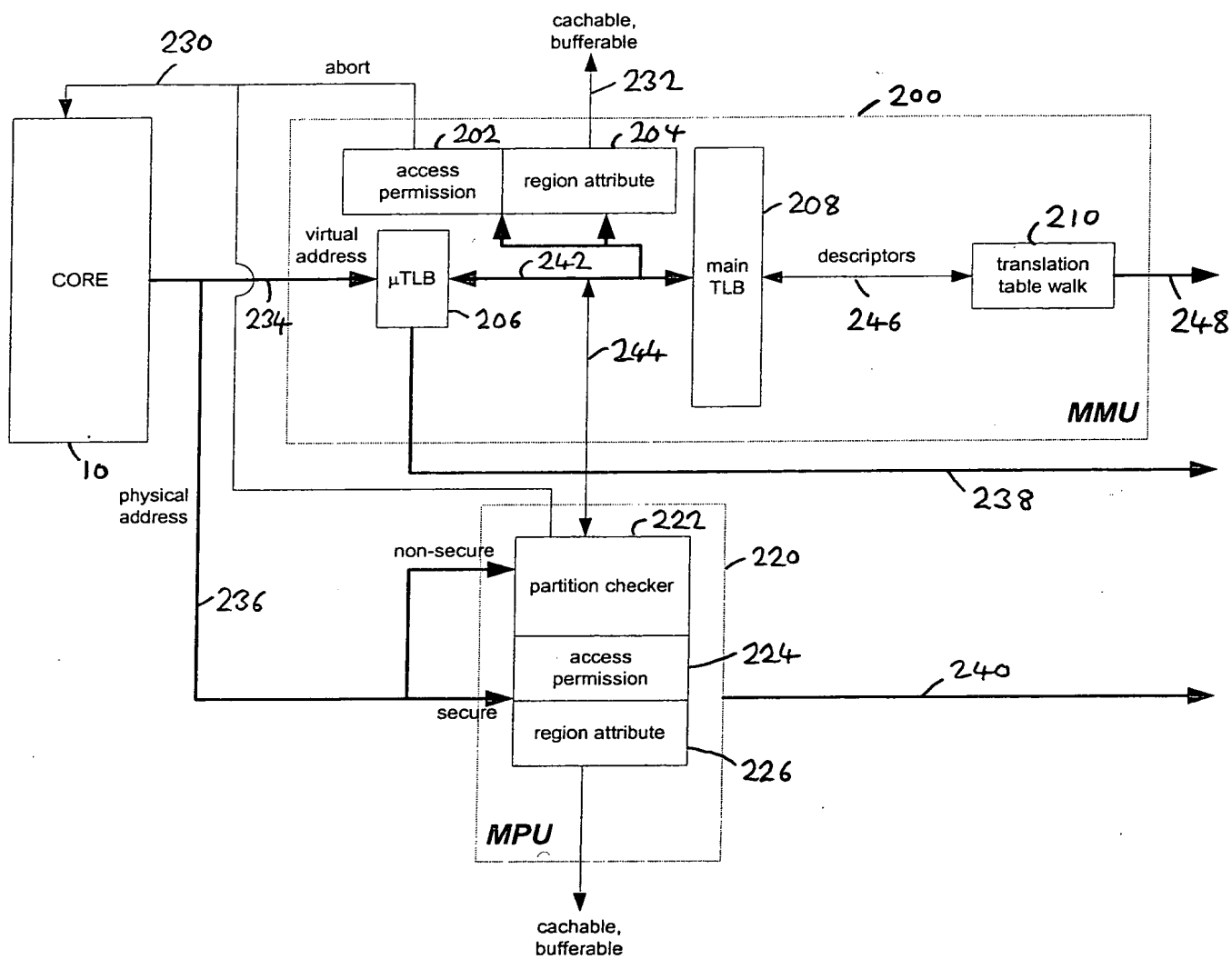


FIG. 21

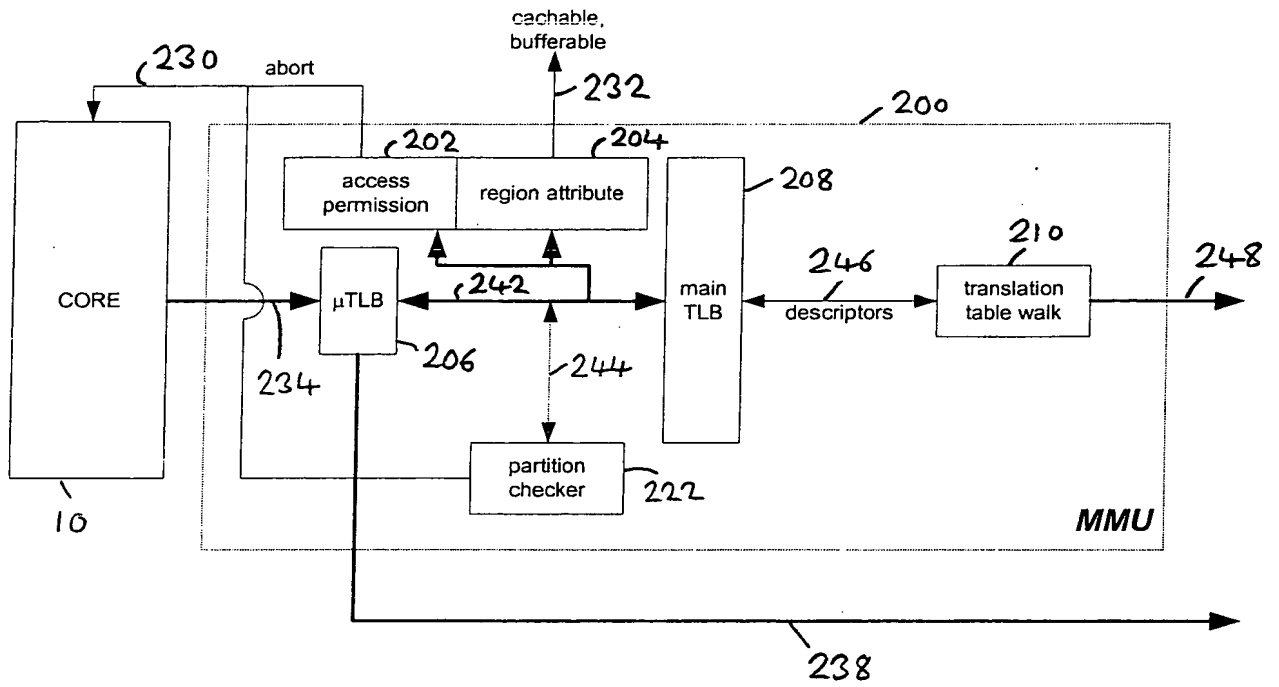
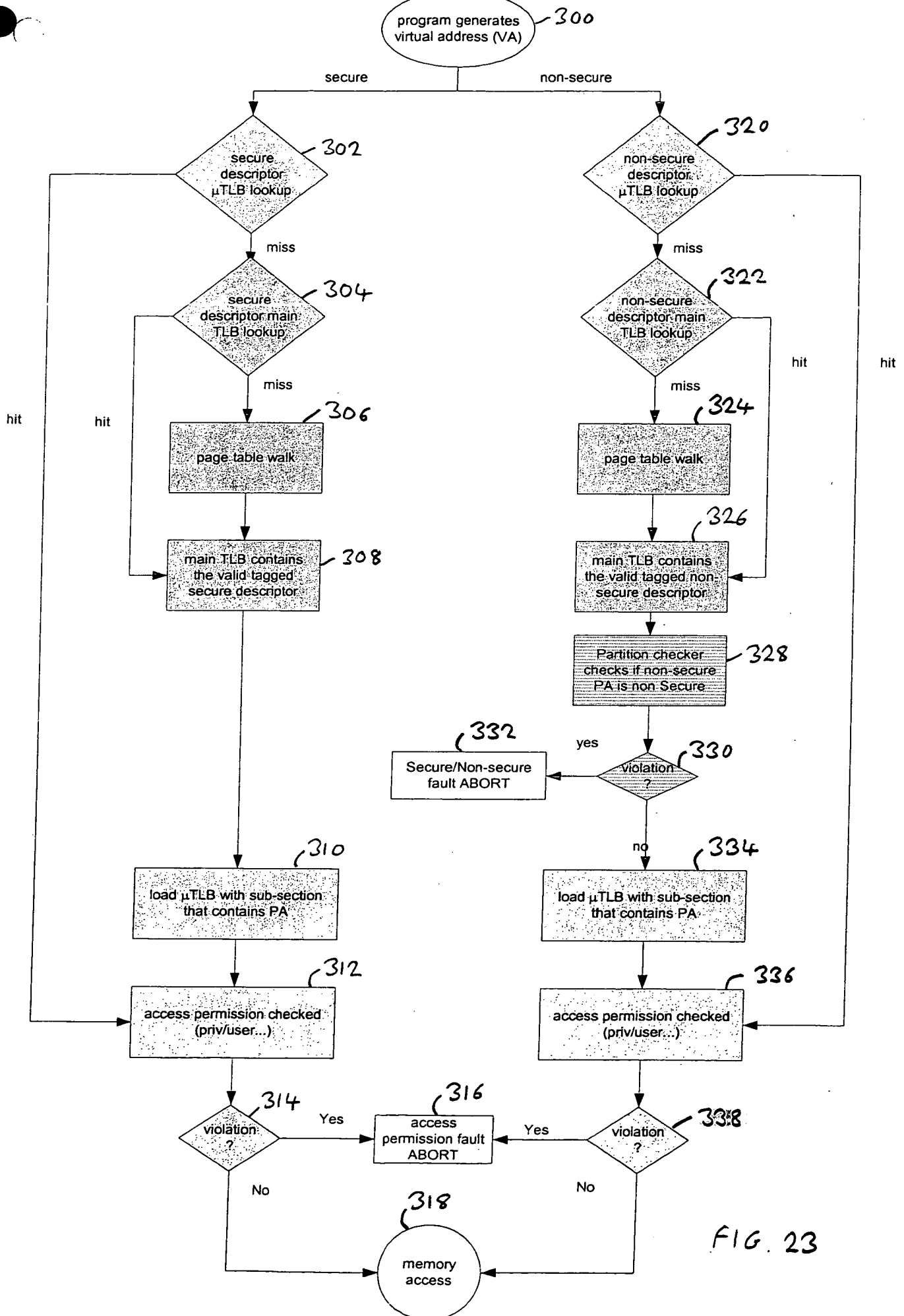


FIG. 22

17/39



18/39

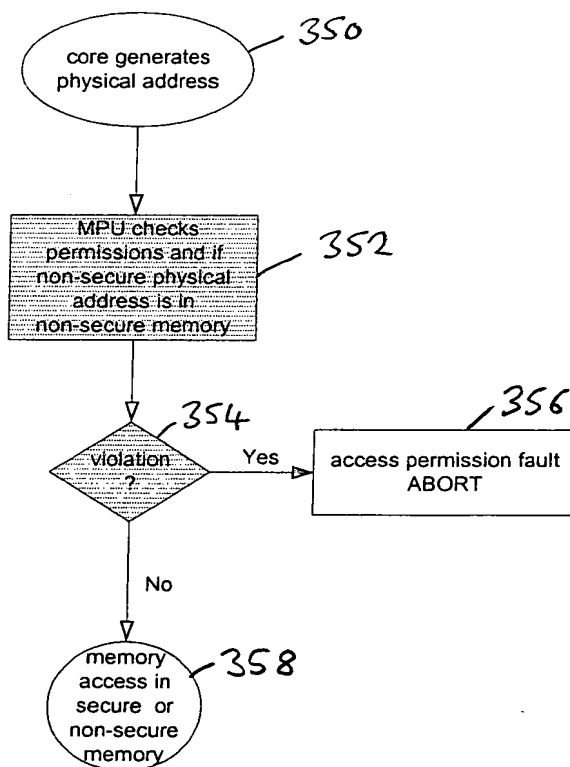
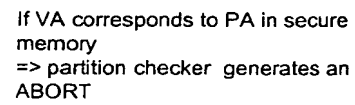


FIG. 24

the descriptor
describes a section
that overlaps
Secure regions



If VA corresponds to PA in non-secure memory
=> this region is loaded in uTLB

FIG. 25

20/39

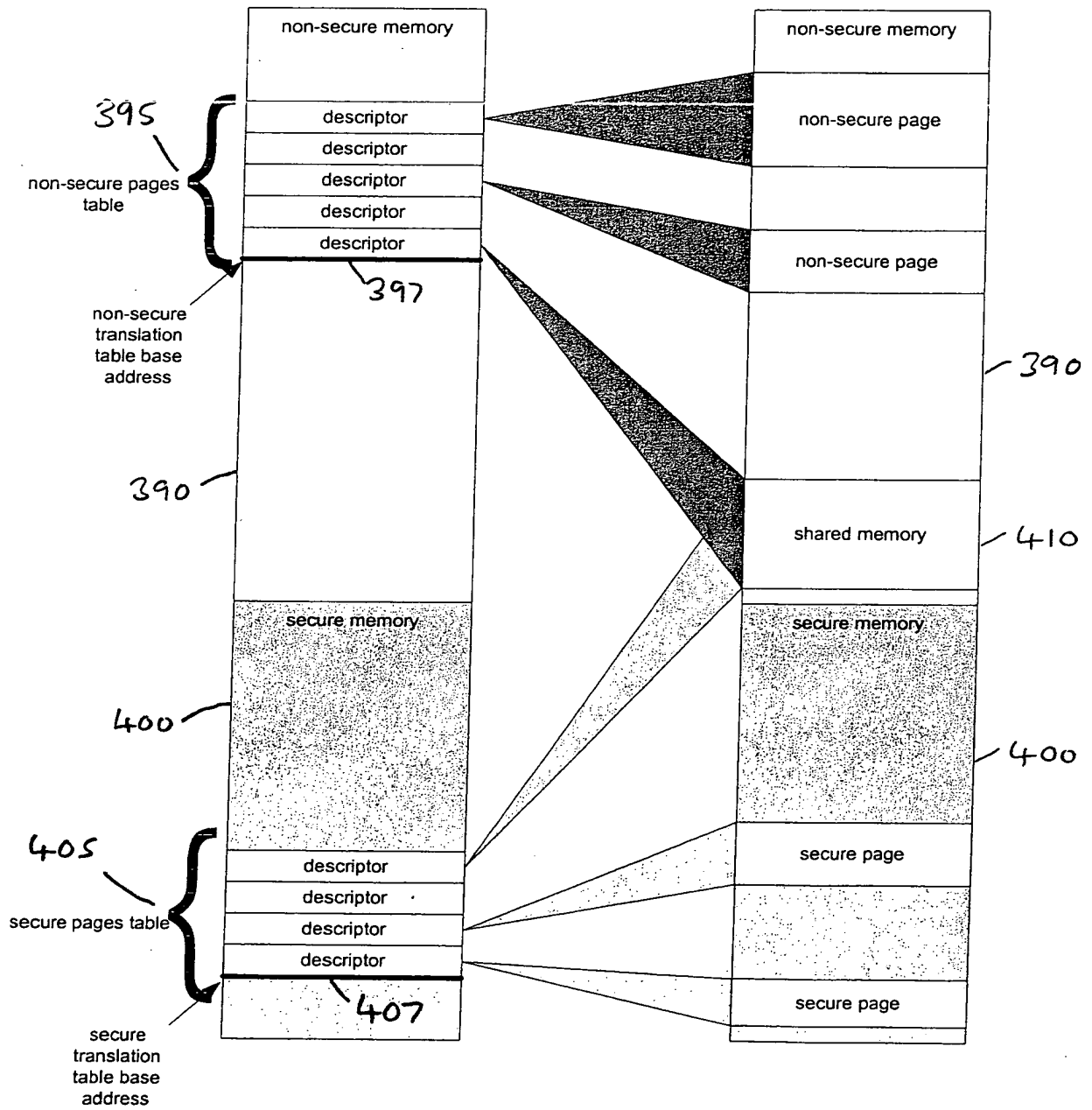


FIG. 26

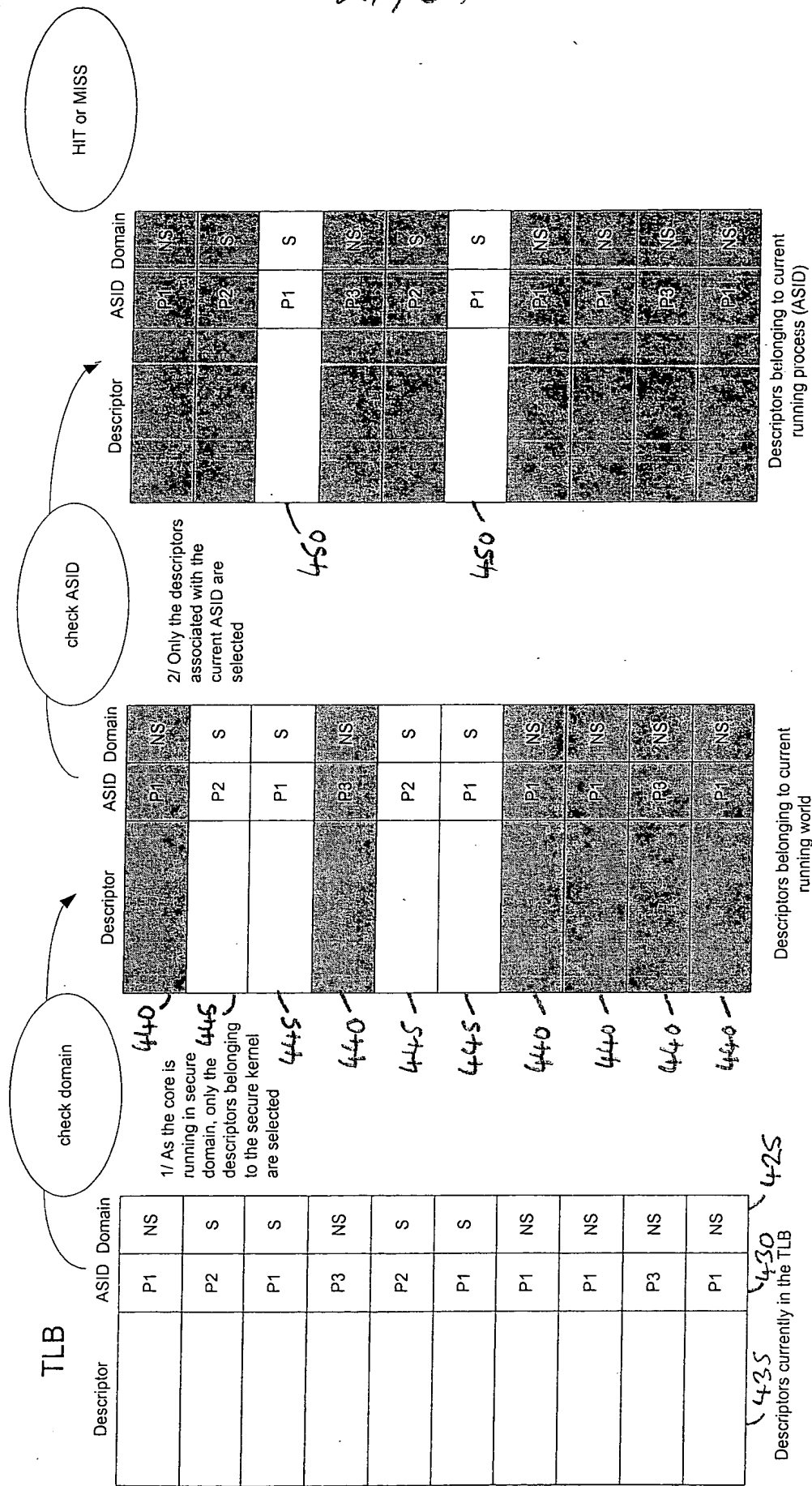


FIG. 27

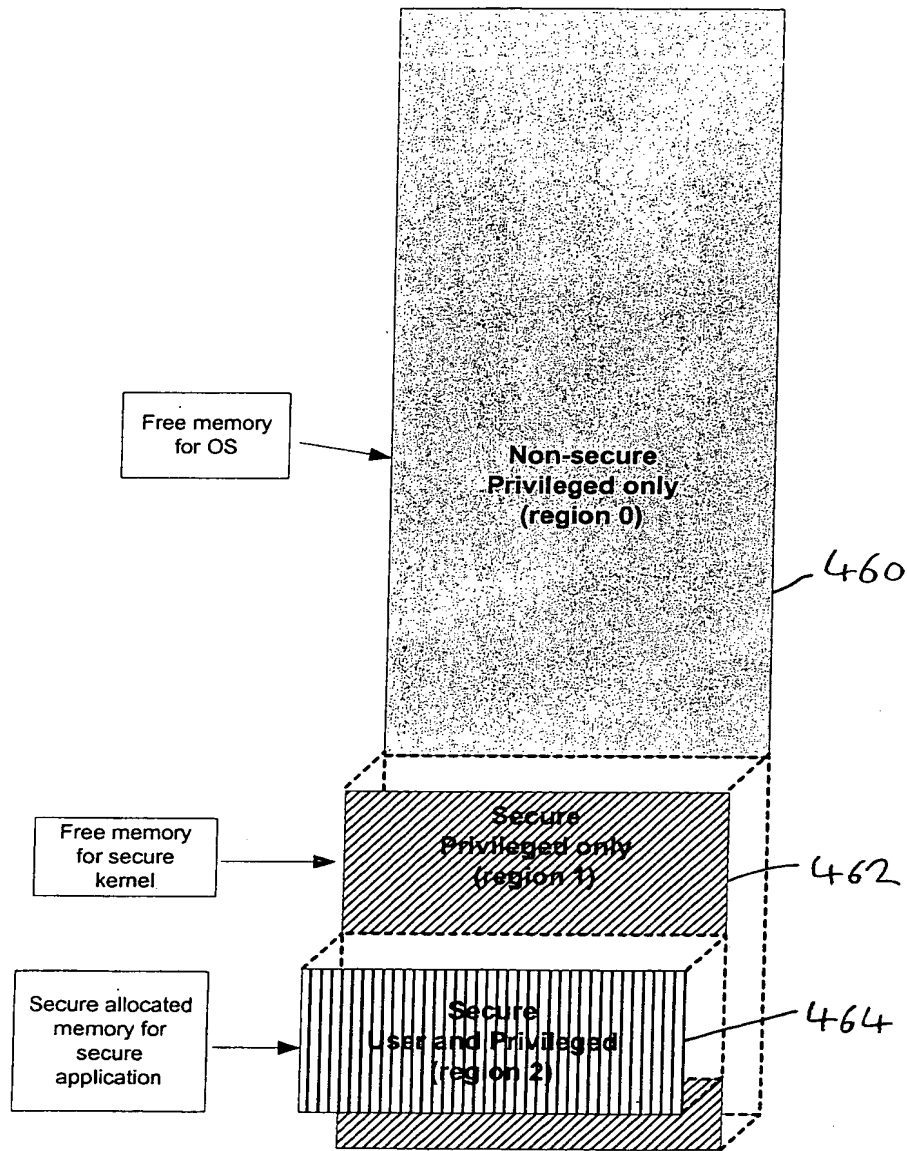


FIG. 28

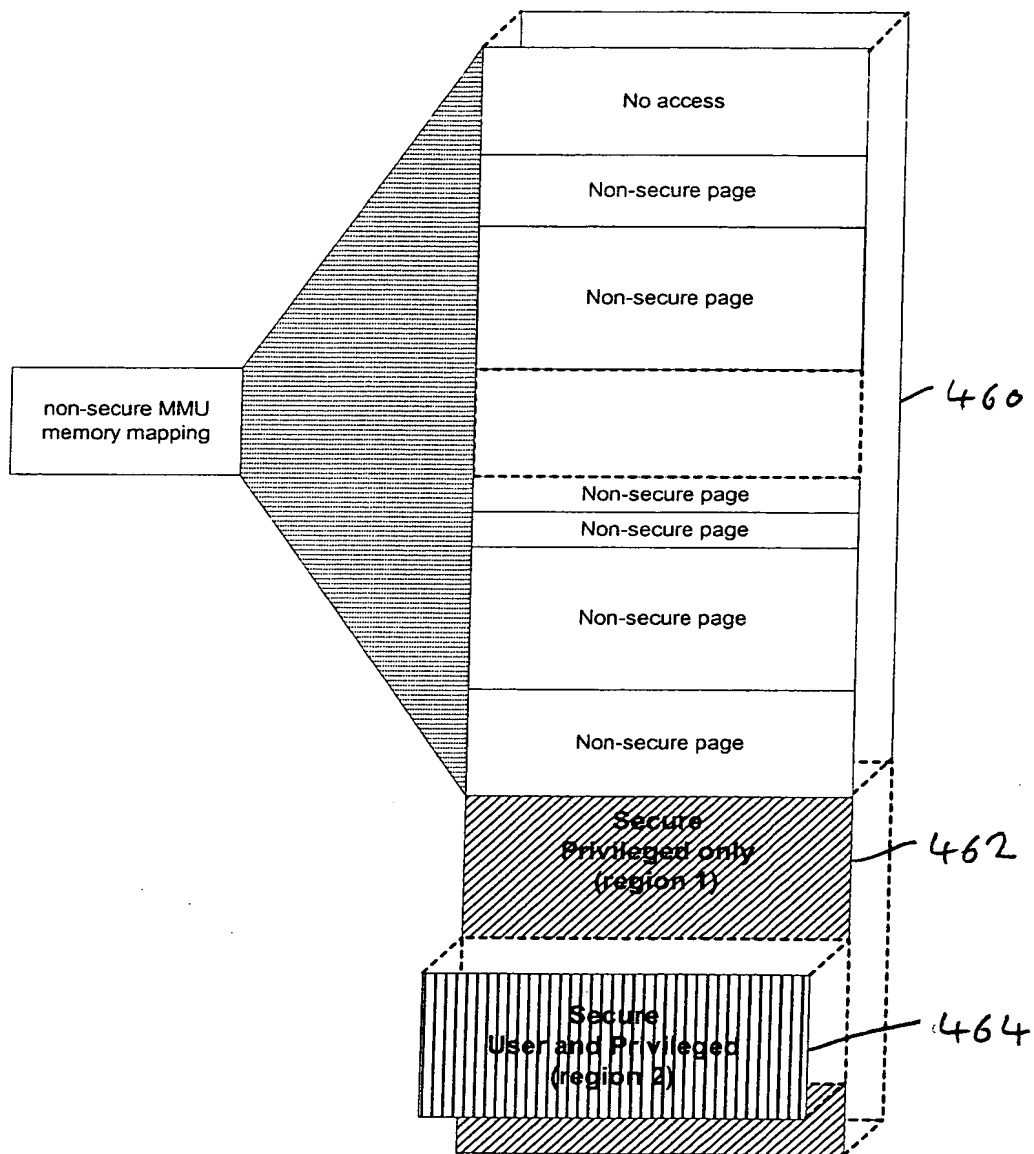


FIG. 29

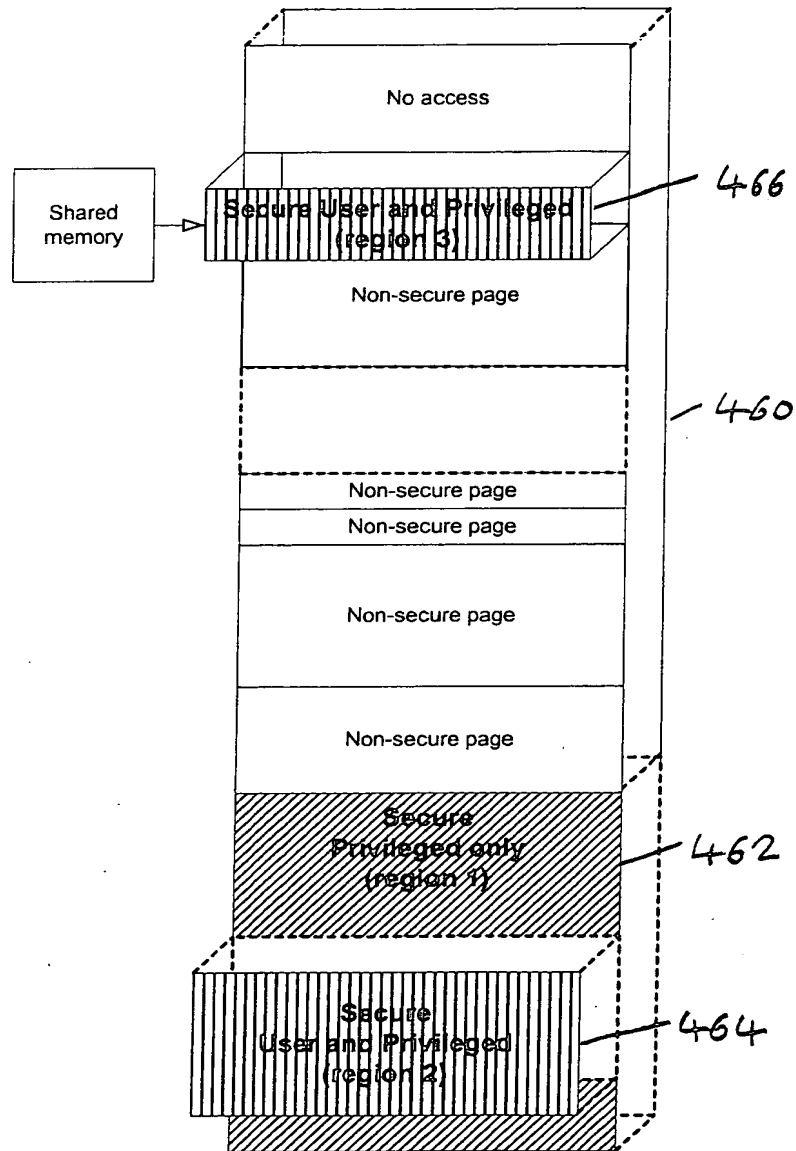


FIG. 30

25/39

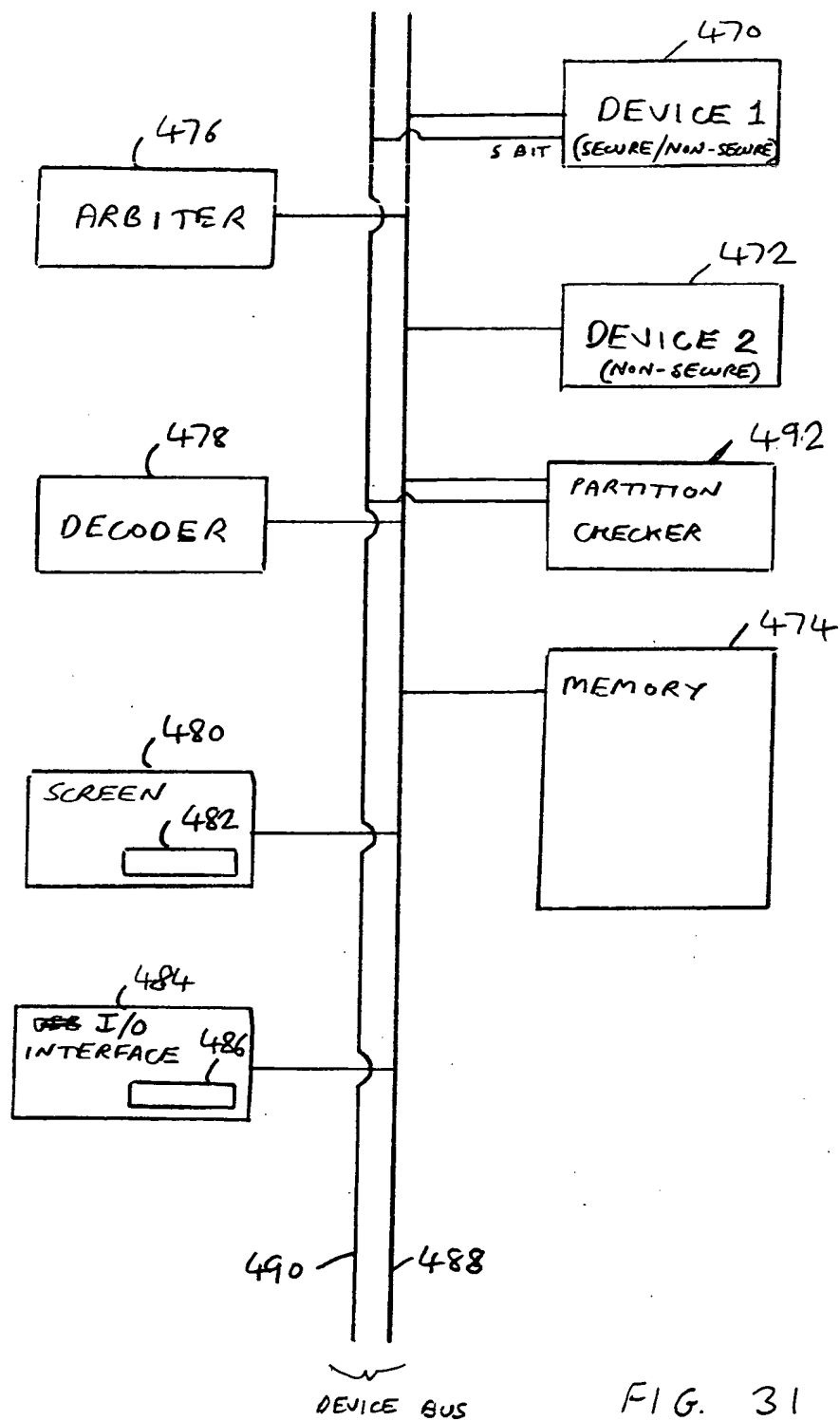


FIG. 31

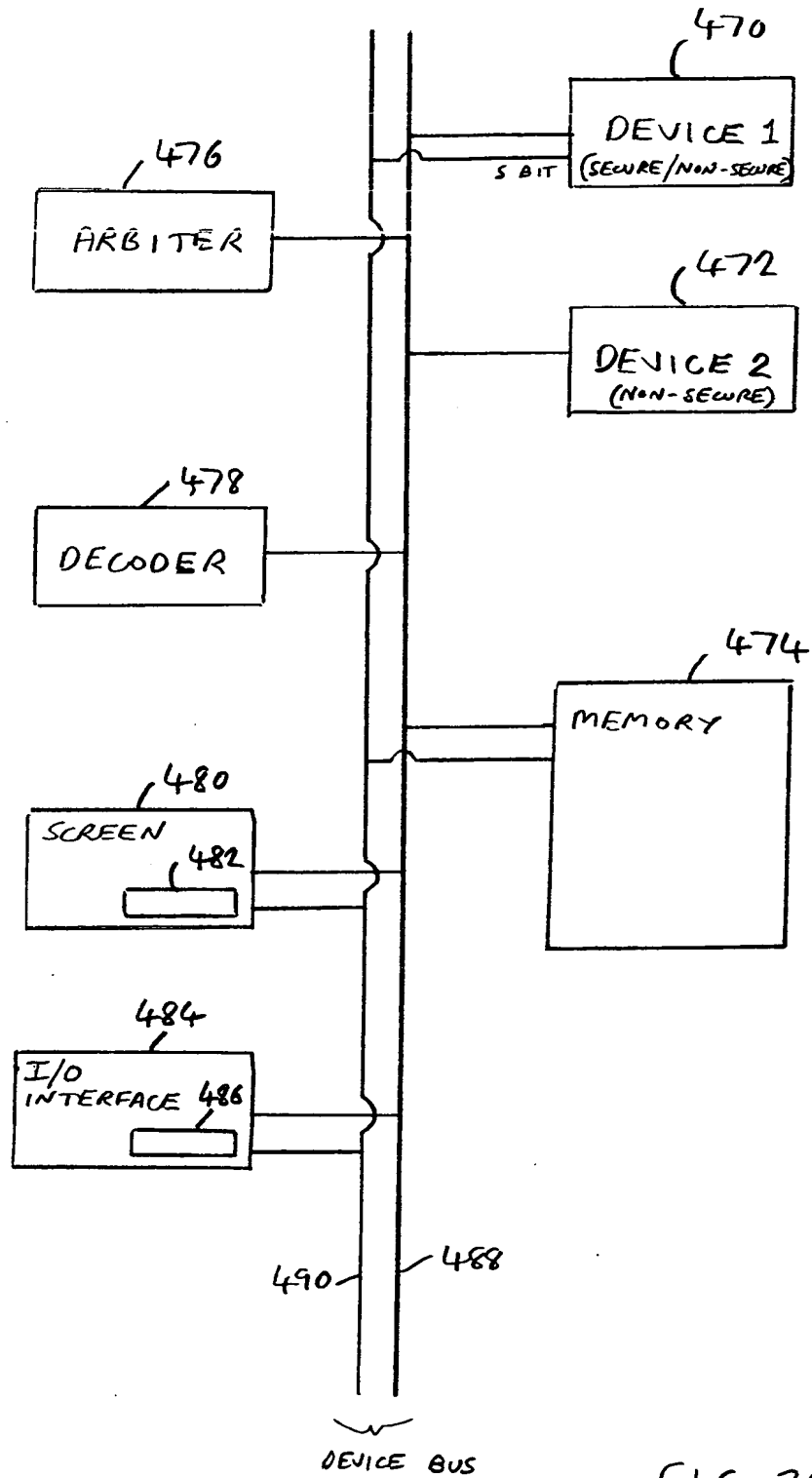


FIG. 32



7 June 33

Method of entry	How to program?	How to enter?	Entry mode
Breakpoint hits	Debug TAP or software (CP14)	Program breakpoint register and/or context-ID register and comparisons succeed with Instruction Address and/or CP15 Context ID ⁽²⁾ .	Halt/monitor ⁽¹⁾
Software breakpoint instruction	Put a BKPT instruction into scan chain 4 (Instruction Transfer Register) through Debug TAP or Use BKPT instruction directly in the code.	BKPT instruction must reach execution stage.	Halt/monitor
Vector trap breakpoint	Debug TAP	Program vector trap register and address matches.	Halt/monitor
Watchpoint hits	Debug TAP or software (CP14)	Program watchpoint register and/or context-ID register and comparisons succeed with Instruction Address and/or CP15 Context ID ⁽²⁾ .	Halt/monitor ⁽¹⁾
Internal debug request	Debug TAP	Halt instruction has been scanned in.	Halt
External debug request	Not applicable	EDBGRQ input pin is asserted.	Halt

⁽¹⁾: In monitor mode, breakpoints and watchpoints cannot be data-dependent.




⁽²⁾: The cores have support for thread-aware breakpoints and watchpoints, in order to enable secure debug on some particular threads.

Figure 34

Name	Meaning	Reset value	Access	Inserted in scan chain for test
Monitor mode enable bit	0: halt mode 1: monitor mode	1	R/W by programming the ICE by the JTAG (scan1) ▪ R/W by using MRC/MCR instruction (CP14)	yes
Secure debug enable bit	0: debug in non-secure world only. 1: debug in secure world and non-secure world	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1)	no
Secure trace enable bit	0: ETM is enabled in non-secure world only. 1: ETM is enabled in secure world and non-secure world	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1)	no
Secure user-mode enable bit	0: debug is not possible in secure user mode 1: debug is possible in secure user mode	1	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1)	no
Secure thread-aware enable bit	0: debug is not possible for a particular thread 1: debug is possible for a particular thread	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1)	no

Figure 35

Function Table

D	CK	Q[n+1]
0		0
1		1
X		Q[n]

Logic Symbol

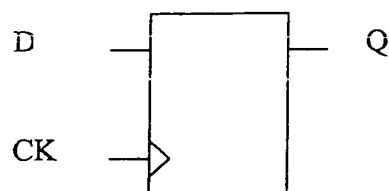







FIGURE 36

Function Table

D	SI	SE	CK	Q[n+1]
0	X	0		0
1	X	0		1
X	X	X		Q[n]
X	0	1		0
X	1	1		1

Logic Symbol

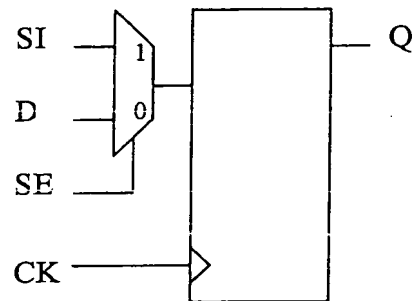


Figure 37

32/39

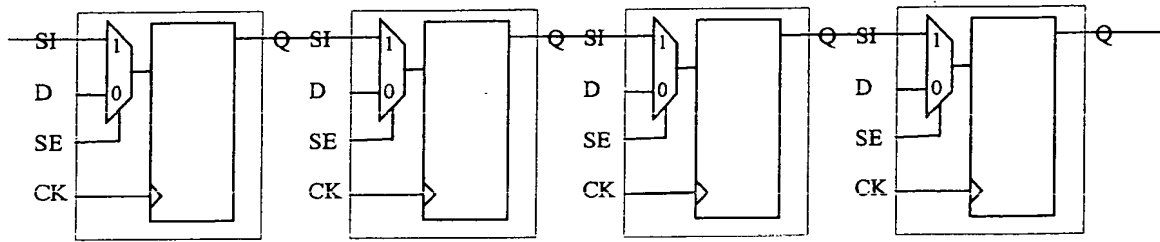


FIGURE 38

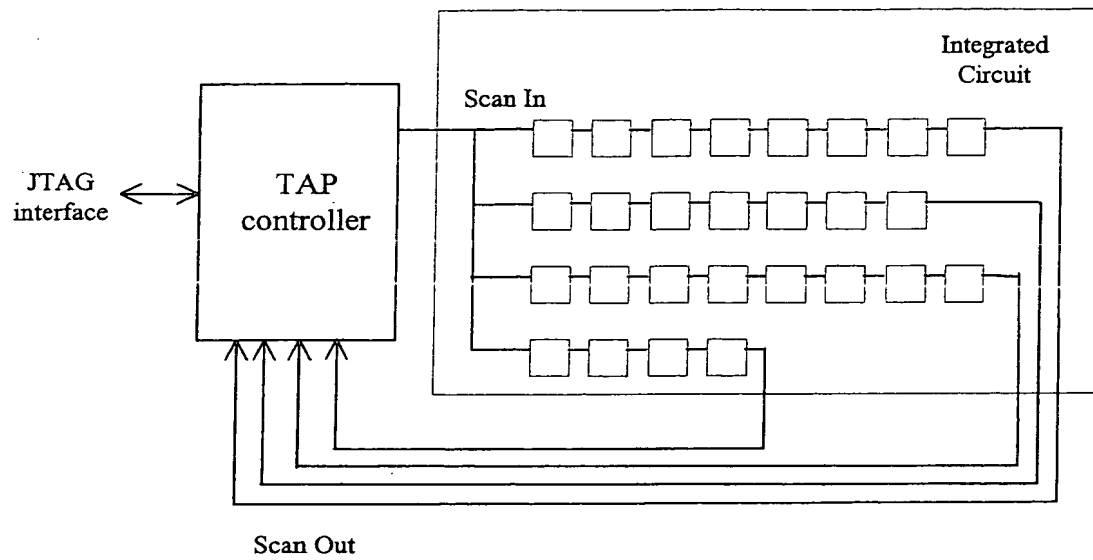


FIGURE 39

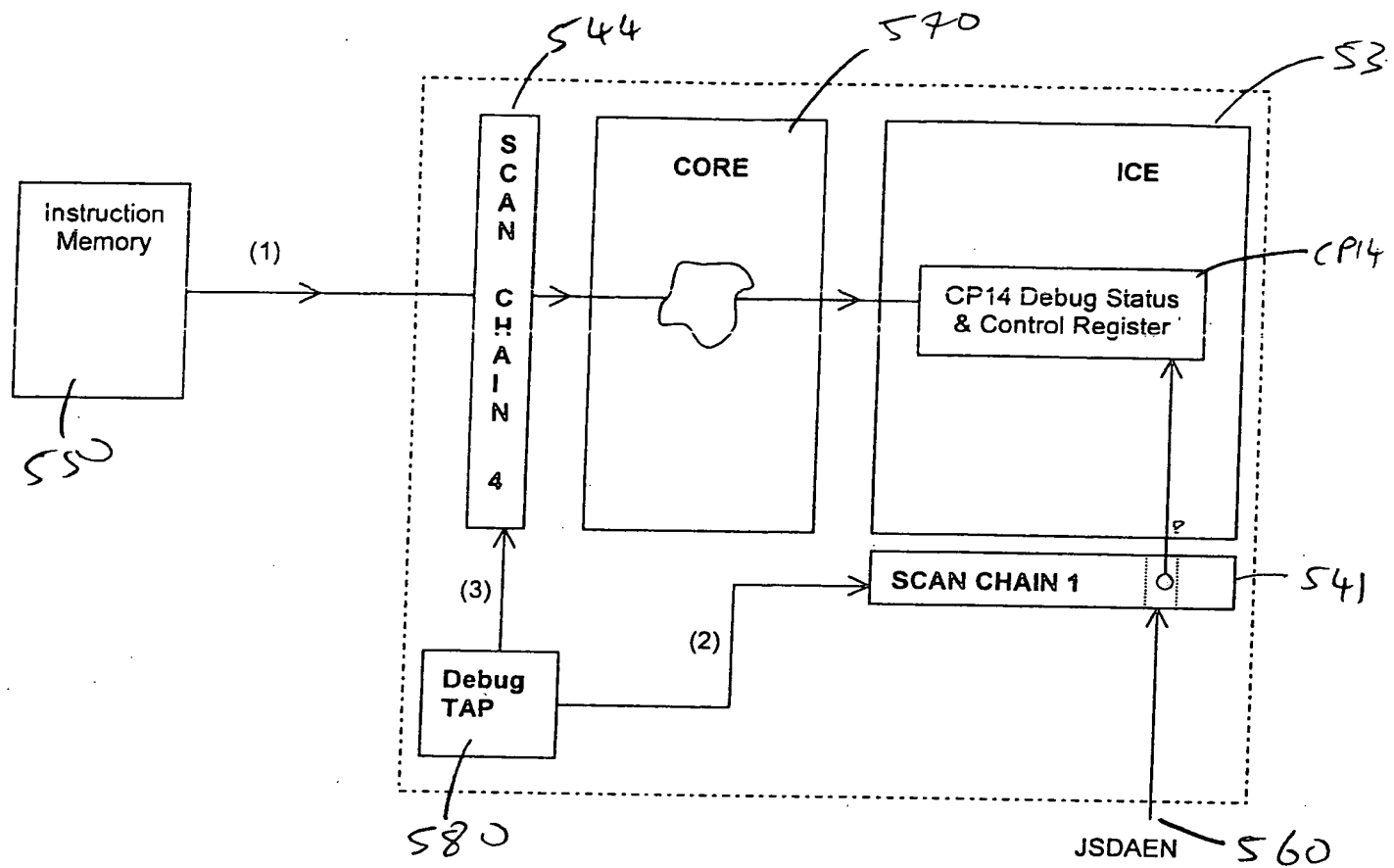


Figure 41

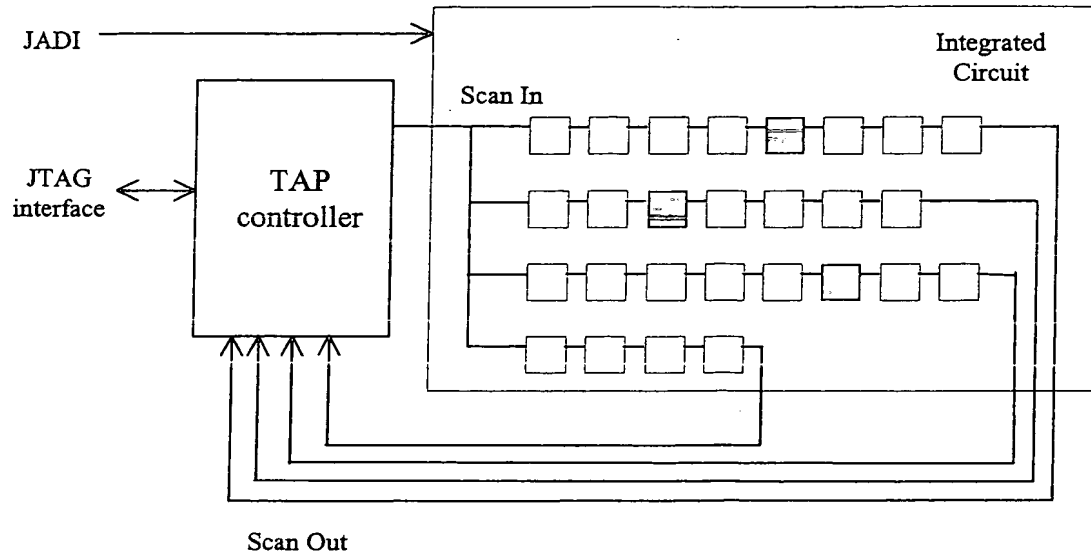


FIGURE 40A

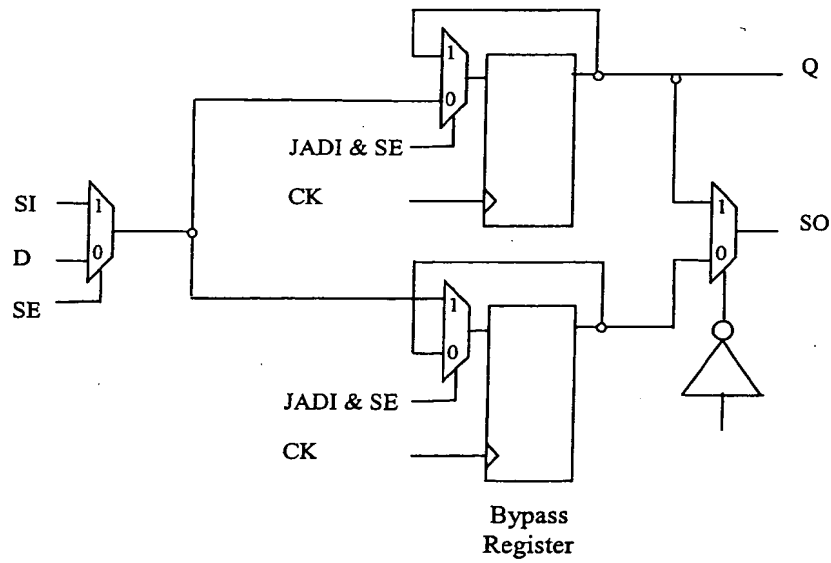


FIGURE 40B

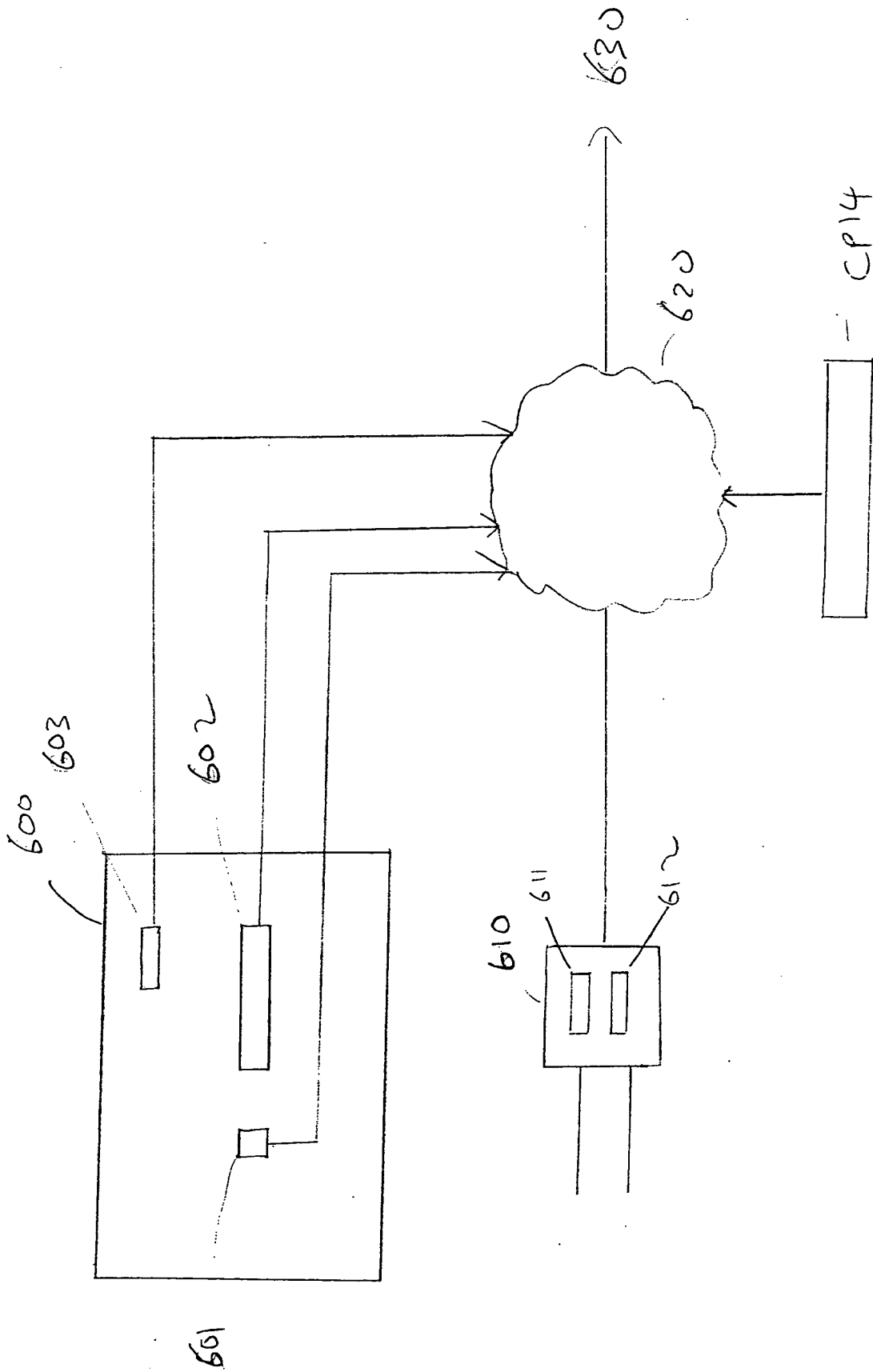


Figure 42

CP14 bits in Debug and Status Control register			meaning
Secure debug enable bit	Secure user-mode debug enable bit	Secure thread-aware debug enable bit	
0	X	X	No intrusive debug in entire secure world is possible. Any debug request, breakpoints, watchpoints, and other mechanism to enter debug state are ignored in entire secure world.
1	0	X	Debug in entire secure world is possible
1	1	0	Debug in secure user-mode only. Any debug request, breakpoints, watchpoints, and other mechanism to enter debug state are taken into account in user mode only. (Breakpoints and watchpoints linked or not to a thread ID are taken into account). Access in debug is restricted to what secure user can have access to.
1	1	1	Debug is possible only in some particular threads. In that case only thread-aware breakpoints and watchpoints linked to a thread ID are taken into account to enter debug state. Each thread can moreover debug its own code, and only its own code.

Figure 43A

CP14 bits in Debug and Status Control register			meaning
Secure trace enable bit	Secure user-mode debug enable bit	Secure thread-aware debug enable bit	
0	X	X	No observable debug in entire secure world is possible. Trace module (ETM) must not trace internal core activity.
1	0	X	Trace in entire secure world is possible
1	1	0	Trace is possible when the core is in secure user-mode only.
1	1	1	Trace is possible only when the core is executing some particular threads in secure user mode. Particular hardware must be dedicated for this, or re-use breakpoint register pair: Context ID match must enable trace instead of entering debug state.

Figure 43B

38/39

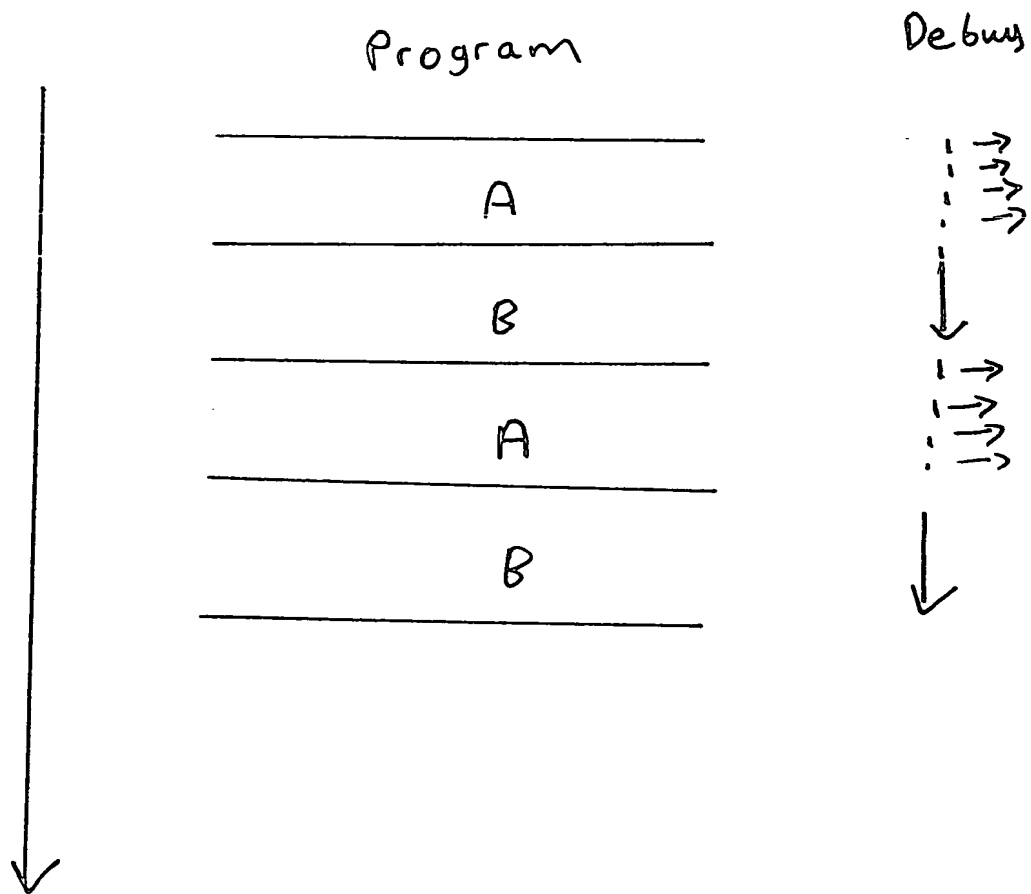


Figure 44

Method of entry	Entry when in non-secure world	entry when in secure world
Breakpoint hits	Non-secure prefetch abort handler	secure prefetch abort handler
Software breakpoint instruction	Non-secure prefetch abort handler	secure prefetch abort handler
Vector trap breakpoint	Disabled for non-secure data abort and non-secure prefetch abort interruptions. For other non-secure exceptions, prefetch abort.	Disabled for secure data abort and secure prefetch abort exceptions ⁽¹⁾ . For other exceptions, secure prefetch abort.
Watchpoint hits	Non-secure data abort handler	secure data abort handler
Internal debug request	Debug state in halt mode	debug state in halt mode
External debug request	Debug state in halt mode	debug state in halt mode

⁽¹⁾ See information on vector trap register.

(2) Note that when external or internal debug request is asserted, the core enters halt mode and not monitor mode.

Figure 45A

Method of entry	Entry in non-secure world	entry in secure world
Breakpoint hits	Non-secure prefetch abort handler	breakpoint ignored
Software breakpoint instruction	Non-secure prefetch abort handler	instruction ignored ⁽¹⁾
Vector trap breakpoint	Disabled for non-secure data abort and non-secure prefetch abort interruptions. For others interruption non-secure prefetch abort.	breakpoint ignored
Watchpoint hits	Non-secure data abort handler	watchpoint ignored
Internal debug request	Debug state in halt mode	request ignored
External debug request	Debug state in halt mode	request ignored
Debug re-entry from system speed access	not applicable	not applicable

⁽¹⁾ As substitution of BKPT instruction in secure world from non-secure world is not possible, non-secure abort must handle the violation.

Figure 45B